deepchem

Release 2.8.1.dev

deepchem-contributors

May 01, 2024

GET STARTED

1 What is DeepChem? 3

2 Quick Start 5

3 About Us 7

Bibliography 825

Index 827

The DeepChem project aims to democratize deep learning for science.

CHAPTER

ONE

WHAT IS DEEPCHEM?

The DeepChem project aims to build high quality tools to democratize the use of deep learning in the sciences. The origin of DeepChem focused on applications of deep learning to chemistry, but the project has slowly evolved past its roots to broader applications of deep learning to the sciences.

The core DeepChem Repo serves as a monorepo that organizes the DeepChem suite of scientific tools. As the project matures, smaller more focused tool will be surfaced in more targeted repos. DeepChem is primarily developed in Python, but we are experimenting with adding support for other languages.

What are some of the things you can use DeepChem to do? Here's a few examples:

- Predict the solubility of small drug-like molecules
- Predict binding affinity for small molecule to protein targets
- Predict physical properties of simple materials
- Analyze protein structures and extract useful descriptors
- Count the number of cells in a microscopy image
- More coming soon…

We should clarify one thing up front though. DeepChem is a machine learning library, so it gives you the tools to solve each of the applications mentioned above yourself. DeepChem may or may not have prebaked models which can solve these problems out of the box.

Over time, we hope to grow the set of scientific applications DeepChem can address. This means we need lots of help! If you're a scientist who's interested in open source, please pitch on building DeepChem.

CHAPTER TWO

QUICK START

The fastest way to get up and running with DeepChem is to run it on Google Colab. Check out one of the DeepChem Tutorials.

If you'd like to install DeepChem locally,

```
pip install deepchem
```

Then open your IDE or text editor of choice and try running the following code with python.

```python
import deepchem
```

CHAPTER THREE

ABOUT US

DeepChem is managed by a team of open source contributors. Anyone is free to join and contribute! DeepChem has weekly developer calls. You can find meeting minutes on our forums.

DeepChem developer calls are open to the public! To listen in, please email X.Y@gmail.com, where X=bharath and Y=ramsundar to introduce yourself and ask for an invite.

Important:

Join our community gitter to discuss DeepChem.
Sign up for our forums to talk about research, development, and general questions.

3.1 Installation

3.1.1 Stable version

Install deepchem via pip or conda by simply running,

```
pip install deepchem
```

or

```
conda install -c conda-forge deepchem
```

3.1.2 Nightly build version

The nightly version is built by the HEAD of DeepChem.

For using general utilites like Molnet, Featurisers, Datasets, etc, then, you install deepchem via pip.

```
pip install --pre deepchem
```

Deepchem provides support for tensorflow, pytorch, jax and each require a individual pip Installation.

For using models with tensorflow dependencies, you install using

```
pip install --pre deepchem[tensorflow]
```

For using models with Pytorch dependencies, you install using

```
pip install --pre deepchem[torch]
```

For using models with Jax dependencies, you install using

```
pip install --pre deepchem[jax]
```

If GPU support is required, then make sure CUDA is installed and then install the desired deep learning framework using the links below before installing deepchem

1. tensorflow - just cuda installed
2. pytorch - https://pytorch.org/get-started/locally/#start-locally
3. jax - https://github.com/google/jax#pip-installation-gpu-cuda

In `zsh` square brackets are used for globbing/pattern matching. This means you need to escape the square brackets in the above installation. You can do so by including the dependencies in quotes like `pip install --pre 'deepchem[jax]'`

Note: Support for jax is not available in windows (jax is not officially supported in windows).

3.1.3 Google Colab

The fastest way to get up and running with DeepChem is to run it on Google Colab. Check out one of the DeepChem Tutorials or this forum post for Colab quick start guides.

3.1.4 Docker

If you want to install using a docker, you can pull two kinds of images from DockerHub.

- **deepchemio/deepchem:x.x.x**
  - Image built by using a conda (x.x.x is a version of deepchem)
  - This image is built when we push x.x.x. tag
  - Dockerfile is put in **`docker/tag`_** directory
- **deepchemio/deepchem:latest**
  - Image built from source codes
  - This image is built every time we commit to the master branch
  - Dockerfile is put in **`docker/nightly`_** directory

First, you pull the image you want to use.

```
docker pull deepchemio/deepchem:latest
```

Then, you create a container based on the image.

```
docker run --rm -it deepchemio/deepchem:latest
```

If you want GPU support:

```
# If nvidia-docker is installed
nvidia-docker run --rm -it deepchemio/deepchem:latest
docker run --runtime nvidia --rm -it deepchemio/deepchem:latest

# If nvidia-container-toolkit is installed
docker run --gpus all --rm -it deepchemio/deepchem:latest
```

You are now in a docker container which deepchem was installed. You can start playing with it in the command line.

```
(deepchem) root@xxxxxxxxxxxxx:~/mydir# python
Python 3.10.13 |Anaconda, Inc.| (default, Aug 24 2023, 12:59:26)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import deepchem as dc
```

If you want to check the tox21 benchmark:

```
# you can run our tox21 benchmark
(deepchem) root@xxxxxxxxxxxxx:~/mydir# wget https://raw.githubusercontent.com/deepchem/
↪deepchem/master/examples/benchmark.py
(deepchem) root@xxxxxxxxxxxxx:~/mydir# python benchmark.py -d tox21 -m graphconv -s␣
↪random
```

3.1.5 Jupyter Notebook

Installing via these steps will allow you to install and import DeepChem into a jupyter notebook within a conda virtual environment.

Prerequisite

- Shell: Bash, Zsh, PowerShell
- Conda: >4.6

First, please create a conda virtual environment (here it's named "deepchem-test") and activate it.

```
conda create --name deepchem-test
conda activate deepchem-test
```

Install DeepChem, Jupyter and matplotlib into the conda environment.

```
conda install -y -c conda-forge nb_conda_kernels matplotlib
pip install tensorflow
pip install --pre deepchem
```

You may need to use `pip3` depending on your Python 3 pip installation. Install pip dependencies after deepchem-test is activated.

While the deepchem-test environment is activated, open Jupyter Notebook by running `jupyter notebook`. Your terminal prompt should be prefixed with (deepchem-test). Once Jupyter Notebook opens in a browser, select the new button, and select the environment "Python[conda env:deepchem-test]." This will open a notebook running in the deepchem-test conda virtual environment.

3.1.6 From source with conda

Installing via these steps will ensure you are installing from the source.

Prerequisite

- Shell: Bash, Zsh, PowerShell
- Conda: >4.6

First, please clone the deepchem repository from GitHub.

```
git clone https://github.com/deepchem/deepchem.git
cd deepchem
```

Then, execute the shell script. The shell scripts require two arguments, **python version** and **gpu/cpu**.

```
source scripts/install_deepchem_conda.sh 3.10 cpu
```

If you want GPU support (we supports only CUDA 11.8):

```
source scripts/install_deepchem_conda.sh 3.10 gpu
```

If you are using the Windows and the PowerShell:

```
.\scripts\install_deepchem_conda.ps1 3.10 cpu
```

Sometimes, PowerShell scripts can't be executed due to problems in Execution Policies.
In that case, you can either change the Execution policies or use the bypass argument.

```
powershell -executionpolicy bypass -File .\scripts\install_deepchem_conda.ps1 3.10 cpu
```

Before activating deepchem environment, make sure conda has been initialized.
Check if there is a `(XXXX)` in your command line.
If not, use `conda init <YOUR_SHELL_NAME>` to activate it, then:

```
conda activate deepchem
pip install -e .
pytest -m "not slow" deepchem # optional
```

3.1.7 From source lightweight guide

Installing via these steps will ensure you are installing from the source.

Prerequisite

- Shell: Bash, Zsh, PowerShell
- Conda: >4.6

First, please clone the deepchem repository from GitHub.

```
git clone https://github.com/deepchem/deepchem.git
cd deepchem
```

We would advise all users to use conda environment, following below-

```
conda create --name deepchem python=3.10
conda activate deepchem
pip install -e .
```

DeepChem provides diffrent additional packages depending on usage & contribution If one also wants to build the tensorflow environment, add this

```
pip install -e .[tensorflow]
```

If one also wants to build the Pytorch environment, add this

```
pip install -e .[torch]
```

If one also wants to build the Jax environment, add this

```
pip install -e .[jax]
```

DeepChem has soft requirements, which can be installed on the fly during development inside the environment but if you want to install all the soft-dependencies at once, then take a look at deepchem/requirements

3.2 Requirements

3.2.1 Hard requirements

DeepChem officially supports Python 3.8 through 3.10 and requires these packages on any condition.

- joblib
- NumPy
- pandas
- scikit-learn
- SymPy
- SciPy

3.2.2 Soft requirements

DeepChem has a number of “soft” requirements.

Package name	Version	Location where this package is used (dc: deepchem)
BioPython	latest	`dc.utlis.genomics_utils`
Deep Graph Library	0.5.x	`dc.feat.graph_data`, `dc.models.torch_models`
DGL-LifeSci	0.2.x	`dc.models.torch_models`
HuggingFace Transformers	Not Testing	`dc.feat.smiles_tokenizer`
HuggingFace Tokenizers	latest	`dc.feat.HuggingFaceVocabularyBuilder`
LightGBM	latest	`dc.models.gbdt_models`
matminer	latest	`dc.feat.materials_featurizers`
MDTraj	latest	`dc.utils.pdbqt_utils`
Mol2vec	latest	`dc.utils.molecule_featurizers`
Mordred	latest	`dc.utils.molecule_featurizers`
NetworkX	latest	`dc.utils.rdkit_utils`
OpenAI Gym	Not Testing	`dc.rl`
OpenMM	latest	`dc.utils.rdkit_utils`
PDBFixer	latest	`dc.utils.rdkit_utils`
Pillow	latest	`dc.data.data_loader`, `dc.trans.transformers`
PubChemPy	latest	`dc.feat.molecule_featurizers`
pyGPGO	latest	`dc.hyper.gaussian_process`
Pymatgen	latest	`dc.feat.materials_featurizers`
PyTorch	2.2.1	`dc.models.torch_models`
PyTorch Geometric	latest (with PyTorch 2.2.1)	`dc.feat.graph_data` `dc.models.torch_models`
RDKit	latest	Many modules (we recommend you to install)
simdna	latest	`dc.metrics.genomic_metrics`, `dc.molnet.dnasim`
TensorFlow	2.15	`dc.models` *deepchem>=2.4.0* depends on TensorFlow v2(2.3.x) *deepchem<2.4.0* depends on TensorFlow v1(>=1.14)
Tensorflow Probability	0.23.x	`dc.rl`
Weights & Biases	Not Testing	`dc.models.keras_model`, `dc.models.callbacks`
XGBoost	latest	`dc.models.gbdt_models`
Tensorflow Addons	latest	`dc.models.optimizers`
pySCF	latest	`dc.models.torch_models.ferminet`
pysam	latest	`dc.feat.bio_seq_featurizer` `dc.models.data_loader`

3.3 Tutorials

If you're new to DeepChem, you probably want to know the basics. What is DeepChem? Why should you care about using it? The short answer is that DeepChem is a scientific machine learning library. (The "Chem" indicates the historical fact that DeepChem initially focused on chemical applications, but we aim to support all types of scientific applications more broadly).

Why would you want to use DeepChem instead of another machine learning library? Simply put, DeepChem maintains an extensive collection of utilities to enable scientific deep learning including classes for loading scientific datasets, processing them, transforming them, splitting them up, and learning from them. Behind the scenes DeepChem uses a variety of other machine learning frameworks such as scikit-learn, TensorFlow, and XGBoost. We are also experimenting with adding additional models implemented in PyTorch and JAX. Our focus is to facilitate scientific experimentation using whatever tools are available at hand.

In the rest of this tutorials, we'll provide a rapid fire overview of DeepChem's API. DeepChem is a big library so we won't cover everything, but we should give you enough to get started.

Contents

- *Data Handling*
- *Feature Engineering*
- *Data Splitting*
- *Model Training and Evaluating*
- *More Tutorials*

3.3.1 Data Handling

The `dc.data` module contains utilities to handle `Dataset` objects. These `Dataset` objects are the heart of DeepChem. A `Dataset` is an abstraction of a dataset in machine learning. That is, a collection of features, labels, weights, alongside associated identifiers. Rather than explaining further, we'll just show you.

```
>>> import deepchem as dc
>>> import numpy as np
>>> N_samples = 50
>>> n_features = 10
>>> X = np.random.rand(N_samples, n_features)
>>> y = np.random.rand(N_samples)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> dataset.X.shape
(50, 10)
>>> dataset.y.shape
(50,)
```

Here we've used the `NumpyDataset` class which stores datasets in memory. This works fine for smaller datasets and is very convenient for experimentation, but is less convenient for larger datasets. For that we have the `DiskDataset` class.

```
>>> dataset = dc.data.DiskDataset.from_numpy(X, y)
>>> dataset.X.shape
(50, 10)
>>> dataset.y.shape
(50,)
```

In this example we haven't specified a data directory, so this `DiskDataset` is written to a temporary folder. Note that `dataset.X` and `dataset.y` load data from disk underneath the hood! So this can get very expensive for larger datasets.

3.3.2 Feature Engineering

"Featurizer" is a chunk of code which transforms raw input data into a processed form suitable for machine learning. The `dc.feat` module contains an extensive collection of featurizers for molecules, molecular complexes and inorganic crystals. We'll show you the example about the usage of featurizers.

```
>>> smiles = [
...   'O=Cc1ccc(O)c(OC)c1',
...   'CN1CCC[C@H]1c2cccnc2',
...   'C1CCCCC1',
...   'c1ccccc1',
...   'CC(=O)O',
... ]
>>> properties = [0.4, -1.5, 3.2, -0.2, 1.7]
>>> featurizer = dc.feat.CircularFingerprint(size=1024)
>>> ecfp = featurizer.featurize(smiles)
>>> ecfp.shape
(5, 1024)
>>> dataset = dc.data.NumpyDataset(X=ecfp, y=np.array(properties))
>>> len(dataset)
5
```

Here, we've used the `CircularFingerprint` and converted SMILES to ECFP. The ECFP is a fingerprint which is a bit vector made by chemical structure information and we can use it as the input for various models.

And then, you may have a CSV file which contains SMILES and property like HOMO-LUMO gap. In such a case, by using `DataLoader`, you can load and featurize your data at once.

```
>>> import pandas as pd
>>> # make a dataframe object for creating a CSV file
>>> df = pd.DataFrame(list(zip(smiles, properties)), columns=["SMILES", "property"])
>>> import tempfile
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...   # dump the CSV file
...   df.to_csv(tmpfile.name)
...   # initizalize the featurizer
...   featurizer = dc.feat.CircularFingerprint(size=1024)
...   # initizalize the dataloader
...   loader = dc.data.CSVLoader(["property"], feature_field="SMILES",␣
↪featurizer=featurizer)
...   # load and featurize the data from the CSV file
...   dataset = loader.create_dataset(tmpfile.name)
...   len(dataset)
5
```

3.3.3 Data Splitting

The `dc.splits` module contains a collection of scientifically aware splitters. Generally, we need to split the original data to training, validation and test data in order to tune the model and evaluate the model's performance. We'll show you the example about the usage of splitters.

```
>>> splitter = dc.splits.RandomSplitter()
>>> # split 5 datapoints in the ratio of train:valid:test = 3:1:1
>>> train_dataset, valid_dataset, test_dataset = splitter.train_valid_test_split(
...   dataset=dataset, frac_train=0.6, frac_valid=0.2, frac_test=0.2
... )
>>> len(train_dataset)
3
>>> len(valid_dataset)
1
>>> len(test_dataset)
1
```

Here, we've used the `RandomSplitter` and splitted the data randomly in the ratio of train:valid:test = 3:1:1. But, the random splitting sometimes overestimates model's performance, especially for small data or imbalance data. Please be careful for model evaluation. The `dc.splits` provides more methods and algorithms to evaluate the model's performance appropriately, like cross validation or splitting using molecular scaffolds.

3.3.4 Model Training and Evaluating

The `dc.models` contains an extensive collection of models for scientific applications. Most of all models inherits `dc.models.Model` and we can train them by just calling `fit` method. You don't need to care about how to use specific framework APIs. We'll show you the example about the usage of models.

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> rf = RandomForestRegressor()
>>> model = dc.models.SklearnModel(model=rf)
>>> # model training
>>> model.fit(train_dataset)
>>> valid_preds = model.predict(valid_dataset)
>>> valid_preds.shape
(1,)
>>> test_preds = model.predict(test_dataset)
>>> test_preds.shape
(1,)
```

Here, we've used the `SklearnModel` and trained the model. Even if you want to train a deep learning model which is implemented by TensorFlow or PyTorch, calling `fit` method is all you need!

And then, if you use `dc.metrics.Metric`, you can evaluate your model by just calling `evaluate` method.

```
>>> # initialze the metric
>>> metric = dc.metrics.Metric(dc.metrics.mae_score)
>>> # evaluate the model
>>> train_score = model.evaluate(train_dataset, [metric])
>>> valid_score = model.evaluate(valid_dataset, [metric])
>>> test_score = model.evaluate(test_dataset, [metric])
```

3.3.5 More Tutorials

DeepChem maintains an extensive collection of addition tutorials that are meant to be run on Google Colab, an online platform that allows you to execute Jupyter notebooks. Once you've finished this introductory tutorial, we recommend working through these more involved tutorials.

3.4 Examples

We show a bunch of examples for DeepChem by the doctest style.

- We match against doctest's `...` wildcard on code where output is usually ignored
- We often use threshold assertions (e.g: `score['mean-pearson_r2_score'] > 0.92`), as this is what matters for model training code.

Contents

- *Delaney (ESOL)*
  - *MultitaskRegressor*
  - *GraphConvModel*
- *ChEMBL*
  - *MultitaskRegressor*
  - *GraphConvModel*

Before jumping in to examples, we'll import our libraries and ensure our doctests are reproducible:

```
>>> import numpy as np
>>> import tensorflow as tf
>>> import deepchem as dc
>>> import random
>>>
>>> # Run before every test for reproducibility
>>> def seed_all():
...     np.random.seed(456)
...     tf.random.set_seed(456)
...     random.seed(456)
```

3.4.1 Delaney (ESOL)

Examples of training models on the Delaney (ESOL) dataset included in MoleculeNet.

We'll be using its `smiles` field to train models to predict its experimentally measured solvation energy (`expt`).

MultitaskRegressor

First, we'll load the dataset with *load_delaney()* and fit a *MultitaskRegressor*:

```
>>> seed_all()
>>> # Load dataset with default 'scaffold' splitting
>>> tasks, datasets, transformers = dc.molnet.load_delaney()
>>> tasks
['measured log solubility in mols per litre']
>>> train_dataset, valid_dataset, test_dataset = datasets
>>>
>>> # We want to know the pearson R squared score, averaged across tasks
>>> avg_pearson_r2 = dc.metrics.Metric(dc.metrics.pearson_r2_score, np.mean)
>>>
>>> # We'll train a multitask regressor (fully connected network)
>>> model = dc.models.MultitaskRegressor(
...     len(tasks),
...     n_features=1024,
...     layer_sizes=[500])
>>>
>>> model.fit(train_dataset)
0...
>>>
>>> # We now evaluate our fitted model on our training and validation sets
>>> train_scores = model.evaluate(train_dataset, [avg_pearson_r2], transformers)
>>> assert train_scores['mean-pearson_r2_score'] > 0.7, train_scores
>>>
>>> valid_scores = model.evaluate(valid_dataset, [avg_pearson_r2], transformers)
>>> assert valid_scores['mean-pearson_r2_score'] > 0.3, valid_scores
```

GraphConvModel

The default featurizer for Delaney is ECFP, short for "Extended-connectivity fingerprints." For a *GraphConvModel*, we'll reload our datasets with `featurizer='GraphConv'`:

```
>>> seed_all()
>>> tasks, datasets, transformers = dc.molnet.load_delaney(featurizer='GraphConv')
>>> train_dataset, valid_dataset, test_dataset = datasets
>>>
>>> model = dc.models.GraphConvModel(len(tasks), mode='regression', dropout=0.5)
>>>
>>> model.fit(train_dataset, nb_epoch=30)
0...
>>>
>>> # We now evaluate our fitted model on our training and validation sets
>>> train_scores = model.evaluate(train_dataset, [avg_pearson_r2], transformers)
>>> assert train_scores['mean-pearson_r2_score'] > 0.5, train_scores
>>>
>>> valid_scores = model.evaluate(valid_dataset, [avg_pearson_r2], transformers)
>>> assert valid_scores['mean-pearson_r2_score'] > 0.3, valid_scores
```

3.4.2 ChEMBL

Examples of training models on ChEMBL dataset included in MoleculeNet.

ChEMBL is a manually curated database of bioactive molecules with drug-like properties. It brings together chemical, bioactivity and genomic data to aid the translation of genomic information into effective new drugs.

MultitaskRegressor

```
>>> seed_all()
>>> # Load ChEMBL 5thresh dataset with random splitting
>>> chembl_tasks, datasets, transformers = dc.molnet.load_chembl(
...     shard_size=2000, featurizer="ECFP", set="5thresh", split="random")
>>> train_dataset, valid_dataset, test_dataset = datasets
>>> len(chembl_tasks)
691
>>> f'Compound train/valid/test split: {len(train_dataset)}/{len(valid_dataset)}/
→{len(test_dataset)}'
'Compound train/valid/test split: 19096/2387/2388'
>>>
>>> # We want to know the RMS, averaged across tasks
>>> avg_rms = dc.metrics.Metric(dc.metrics.rms_score, np.mean)
>>>
>>> # Create our model
>>> n_layers = 3
>>> model = dc.models.MultitaskRegressor(
...     len(chembl_tasks),
...     n_features=1024,
...     layer_sizes=[1000] * n_layers,
...     dropouts=[.25] * n_layers,
...     weight_init_stddevs=[.02] * n_layers,
...     bias_init_consts=[1.] * n_layers,
...     learning_rate=.0003,
...     weight_decay_penalty=.0001,
...     batch_size=100)
>>>
>>> model.fit(train_dataset, nb_epoch=5)
0...
>>>
>>> # We now evaluate our fitted model on our training and validation sets
>>> train_scores = model.evaluate(train_dataset, [avg_rms], transformers)
>>> assert train_scores['mean-rms_score'] < 10.00
>>>
>>> valid_scores = model.evaluate(valid_dataset, [avg_rms], transformers)
>>> assert valid_scores['mean-rms_score'] < 10.00
```

GraphConvModel

```
>>> seed_all()
>>> # Load ChEMBL dataset
>>> chembl_tasks, datasets, transformers = dc.molnet.load_chembl(
...     shard_size=2000, featurizer="GraphConv", set="5thresh", split="random")
>>> train_dataset, valid_dataset, test_dataset = datasets
>>>
>>> # RMS, averaged across tasks
>>> avg_rms = dc.metrics.Metric(dc.metrics.rms_score, np.mean)
>>>
>>> model = dc.models.GraphConvModel(
...     len(chembl_tasks), batch_size=128, mode='regression')
>>>
>>> # Fit trained model
>>> model.fit(train_dataset, nb_epoch=5)
0...
>>>
>>> # We now evaluate our fitted model on our training and validation sets
>>> train_scores = model.evaluate(train_dataset, [avg_rms], transformers)
>>> assert train_scores['mean-rms_score'] < 10.00
>>>
>>> valid_scores = model.evaluate(valid_dataset, [avg_rms], transformers)
>>> assert valid_scores['mean-rms_score'] < 10.00
```

3.5 Known Issues & Limitations

3.5.1 Broken features

A small number of Deepchem features are known to be broken. The Deepchem team will either fix or deprecate these broken features. It is impossible to know of every possible bug in a large project like Deepchem, but we hope to save you some headache by listing features that we know are partially or completely broken.

Note: This list is likely to be non-exhaustive. If we missed something, please let us know [here](https://github.com/deepchem/deepchem/issues/2376).

Feature	Deepchem response	Tracker and notes
ANIFeaturizer/ANIModel	Low Priority Likely deprecate	The Deepchem team recommends using TorchANI instead.

3.5.2 Experimental features

Deepchem features usually undergo rigorous code review and testing to ensure that they are ready for production environments. The following Deepchem features have not been thoroughly tested to the level of other Deepchem modules, and could be potentially problematic in production environments.

Note: This list is likely to be non-exhaustive. If we missed something, please let us know [here](https://github.com/deepchem/deepchem/issues/2376).

Feature	Tracker and notes
Mol2 Loading	Needs more testing.
Interaction Fingerprints	Needs more testing.

If you would like to help us address these known issues, please consider contributing to Deepchem!

3.6 Docker Tutorial

Docker is a software used for easy building, testing and deploying of software. Docker creates an isolated workspace called containers which can avoid dependency version clashes making development of software faster. Also, software can be modularized in different containers, which allows it to be tested without impacting other components or the host computer. Containers contain all the dependencies and the user need not worry about required packages

This makes it easy for users to access older version of deepchem via docker and to develop with them.

Docker works with the following layers:

- Images:

Images are the instructions for creating docker containers. It specifies all the packages and their version to be installed fo the application to run. Images for deep chem can found at docker Hub.

- Containers:

Containers are live instances of Images and are lightweight isolated work-spaces(it does not put much workload on your PC), where you can run and devlop on previous deepchem versions.

- Docker engine:

It is the main application that manages, runs and build containers and images. It also provides a means to interact with the docker container after its built and when it is run.

- Registries:

It is a hub or place where docker images can be found. For deepchem, the default registry is the Docker Hub.

For docker installation, visit: https://docs.docker.com/engine/install/

3.6.1 Using deepchem with docker:

To work with deepchem in docker, we first have to pull deepchem images from docker hub. It can be done in the following way.

if latest deepchem version is needed, then:-

```
#if latest:
docker pull deepchemio/deepchem:latest
```

Else if one wants to work with older version, then the following method should be used:-

```
docker pull deepchemio/deepchem:x.x.x
#x.x.x refers to the version number
```

Now, wait for some time until the image gets downloaded. Then we have to create a container using the image. Then, you have to create a container and use it.

```
docker run --rm -it deepchemio/deepchem:x.x.x
#x.x.x refers to the version number
#replace "x.x.x" with "latest" if latest version is used
```

If you want GPU support:

```
# If nvidia-docker is installed
nvidia-docker run --rm -it deepchemio/deepchem:latest
docker run --runtime nvidia --rm -it deepchemio/deepchem:latest

# If nvidia-container-toolkit is installed
docker run --gpus all --rm -it deepchemio/deepchem:latest
```

Now, you have successfully entered the container's bash where you can execute your programs. **To exit the container press "Ctrl+D".** This stops the container and opens host computer's bash.

To view all the containers present, open up a new terminal/bash of the host computer, then:-

```
docker ps -a
```

This gives a containers list like this:

```
CONTAINER ID   IMAGE                                  COMMAND            CREATED        STATUS      ␣
↪PORTS     NAMES
```

Thus you can see all the created container's Names and its details.

Now you can develop code in you host computer(development environment) and test it in a container having specific version of the deepchem(testing environment).

To test the program you have written, you should copy the program to the container. Open a new host computer's terminal:

```
docker cp host-file-path <container-id>:path-in-container
#container ID should be check in a separate terminal
```

Similarly if you want to copy files out from a container, then open a new host computer's terminal:

```
docker cp <container-id>:path-in-container host-file-path
#container ID should be check in a separate terminal
```

3.6.2 Hands-on tutorial

Lets create a simple deepchem script and work it out in the docker container of deepchem 2.4.0.

Let the script be named deepchem.py in the host computer's location: /home/

deepchem.py contents:

```
import deepchem as dc

print(dc.__version__)
```

Step 1: pull deepchem 2.4.0 image and wait for it to be dowloaded

```
$docker pull deepchemio/deepchem:2.4.0
```

Step 2: Create a container

```
$docker run --rm -it deepchemio/deepchem:2.4.0
(deepchem) root@51b1d2665016:~/mydir#
```

Step 3: Open a new terminal/bash and copy deep.py

```
$docker ps -a
CONTAINER ID   IMAGE                        COMMAND       CREATED         STATUS         ␣
↪PORTS     NAMES
51b1d2665016   deepchemio/deepchem:2.4.0   "/bin/bash"   5 seconds ago   Up 4 seconds   ␣
↪          friendly_lehmann
$docker cp /home/deepchem.py 51b1d2665016:/root/mydir/deepchemp.py
```

step 4: return back to the previous terminal in which container is runing

```
(deepchem) root@51b1d2665016:~/mydir#python3 deepchem.py>>output.txt
2022-01-12 15:33:27.967170: I tensorflow/stream_executor/platform/default/dso_loader.
↪cc:48] Successfully opened dynamic library libcudart.so.10.1
```

This should have created a output file in the container having the deepchem version number. The you should copy it back to host container.

step 5: In a new terminal execute the following commands.

```
$docker cp 51b1d2665016:/root/mydir/output.txt ~/output.txt
$cat ~/output.txt
2.4.0
```

Thus you have successfully executed the program in deepchem 2.4.0!!!

3.7 Documentation Tutorial

This page provides instructions on how to build and test DeepChem documentation.

3.7.1 Building the Documentation

To build the docs, you can use the *Makefile* that's been added to this directory. To generate docs in HTML, run the following commands:

```
$ pip install -r requirements.txt
$ make html
# Clean build
$ make clean html
$ open build/html/index.html
```

3.7.2 Testing

To check if the changes to the docs rendered properly, open build/html on a web browser.

If you want to confirm logs in more detail, use the following command:

```
$ make clean html SPHINXOPTS=-vvv
```

If you want to confirm the example tests, run:

```
$ make doctest_examples
```

3.8 Data

DeepChem `dc.data` provides APIs for handling your data.

If your data is stored by the file like CSV and SDF, you can use the **Data Loaders**. The Data Loaders read your data, convert them to features (ex: SMILES to ECFP) and save the features to Dataset class. If your data is python objects like Numpy arrays or Pandas DataFrames, you can use the **Datasets** directly.

Contents

- *Datasets*
  - *NumpyDataset*
  - *DiskDataset*
  - *ImageDataset*
- *Data Loaders*
  - *CSVLoader*
  - *UserCSVLoader*
  - *ImageLoader*

  - *JsonLoader*
  - *SDFLoader*
  - *FASTALoader*
  - *FASTQLoader*
  - *InMemoryLoader*
  - *Density Functional Theory YAML Loader*
  - *SAM Loader*
  - *BAM Loader*
  - *CRAM Loader*
- *Data Classes*
  - *Graph Data*
  - *Density Functional Theory Data*
- *Base Classes (for develop)*
  - *Dataset*
  - *DataLoader*

3.8.1 Datasets

DeepChem `dc.data.Dataset` objects are one of the core building blocks of DeepChem programs. `Dataset` objects hold representations of data for machine learning and are widely used throughout DeepChem.

The goal of the `Dataset` class is to be maximally interoperable with other common representations of machine learning datasets. For this reason we provide interconversion methods mapping from `Dataset` objects to pandas DataFrames, TensorFlow Datasets, and PyTorch datasets.

NumpyDataset

The `dc.data.NumpyDataset` class provides an in-memory implementation of the abstract `Dataset` which stores its data in `numpy.ndarray` objects.

class NumpyDataset(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ids: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *n_tasks: int = 1*)

A Dataset defined by in-memory numpy arrays.

This subclass of *Dataset* stores arrays *X,y,w,ids* in memory as numpy arrays. This makes it very easy to construct *NumpyDataset* objects.

Examples

```
>>> import numpy as np
>>> dataset = NumpyDataset(X=np.random.rand(5, 3), y=np.random.rand(5,), ids=np.
↪arange(5))
```

`__init__`(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ids: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *n_tasks: int = 1*) → None

Initialize this object.

Parameters

- **X** (`np.ndarray`) – Input features. A numpy array of shape *(n_samples,…)*.
- **y** (`np.ndarray, optional (default None)`) – Labels. A numpy array of shape *(n_samples, …)*. Note that each label can have an arbitrary shape.
- **w** (`np.ndarray, optional (default None)`) – Weights. Should either be 1D array of shape *(n_samples,)* or if there's more than one task, of shape *(n_samples, n_tasks)*.
- **ids** (`np.ndarray, optional (default None)`) – Identifiers. A numpy array of shape *(n_samples,)*
- **n_tasks** (`int, default 1`) – Number of learning tasks.

`__len__`() → int

Get the number of elements in the dataset.

`get_shape`() → Tuple[Tuple[int, ...], Tuple[int, ...], Tuple[int, ...], Tuple[int, ...]]

Get the shape of the dataset.

Returns four tuples, giving the shape of the X, y, w, and ids arrays.

`get_task_names`() → ndarray

Get the names of the tasks associated with this dataset.

`property X: ndarray`

Get the X vector for this dataset as a single numpy array.

`property y: ndarray`

Get the y vector for this dataset as a single numpy array.

`property ids: ndarray`

Get the ids vector for this dataset as a single numpy array.

`property w: ndarray`

Get the weight vector for this dataset as a single numpy array.

`iterbatches`(*batch_size: int | None = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over minibatches from the dataset.

Each minibatch is returned as a tuple of four numpy arrays: *(X, y, w, ids)*.

Parameters

- **batch_size** (*int, optional (default None)*) – Number of elements in each batch.
- **epochs** (*int, default 1*) – Number of epochs to walk over dataset.
- **deterministic** (*bool, optional (default False)*) – If True, follow deterministic order.
- **pad_batches** (*bool, optional (default False)*) – If True, pad each batch to *batch_size*.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

itersamples() → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over the samples in the dataset.

Returns
Iterator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

Examples

```
>>> dataset = NumpyDataset(np.ones((2,2)))
>>> for x, y, w, id in dataset.itersamples():
...   print(x.tolist(), y.tolist(), w.tolist(), id)
[1.0, 1.0] [0.0] [0.0] 0
[1.0, 1.0] [0.0] [0.0] 1
```

transform(*transformer: Transformer*, ***args*) → *NumpyDataset*

Construct a new dataset by applying a transformation to every sample in this dataset.

The argument is a function that can be called as follows: >> newx, newy, neww = fn(x, y, w)

It might be called only once with the whole dataset, or multiple times with different subsets of the data. Each time it is called, it should transform the samples and return the transformed data.

Parameters
transformer (*dc.trans.Transformer*) – The transformation to apply to each sample in the dataset

Returns
A newly constructed NumpyDataset object

Return type
NumpyDataset

select(*indices: Sequence[int] | ndarray*, *select_dir: str | None = None*) → *NumpyDataset*

Creates a new dataset from a selection of indices from self.

Parameters

- **indices** (*List[int]*) – List of indices to select.

- **select_dir** (`str, optional (default None)`) – Used to provide same API as *DiskDataset*. Ignored since *NumpyDataset* is purely in-memory.

Returns
A selected NumpyDataset object

Return type
NumpyDataset

make_pytorch_dataset(*epochs: int = 1*, *deterministic: bool = False*, *batch_size: int | None = None*)

Create a torch.utils.data.IterableDataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w, id) containing the data for one batch, or for a single sample if batch_size is None.

Parameters

- **epochs** (`int, default 1`) – The number of times to iterate over the Dataset
- **deterministic** (`bool, default False`) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **batch_size** (`int, optional (default None)`) – The number of samples to return in each batch. If None, each returned value is a single sample.

Returns
torch.utils.data.IterableDataset that iterates over the data in this dataset.

Return type
torch.utils.data.IterableDataset

Note: This method requires PyTorch to be installed.

static from_DiskDataset(*ds: DiskDataset*) → *NumpyDataset*

Convert DiskDataset to NumpyDataset.

Parameters
ds (`DiskDataset`) – DiskDataset to transform to NumpyDataset.

Returns
A new NumpyDataset created from DiskDataset.

Return type
NumpyDataset

to_json(*fname: str*) → None

Dump NumpyDataset to the json file .

Parameters
fname (`str`) – The name of the json file.

static from_json(*fname: str*) → *NumpyDataset*

Create NumpyDataset from the json file.

Parameters
fname (`str`) – The name of the json file.

Returns
A new NumpyDataset created from the json file.

Return type
NumpyDataset

static merge(*datasets: Sequence[Dataset]*) → *NumpyDataset*

Merge multiple NumpyDatasets.

Parameters

datasets (*List[Dataset]*) – List of datasets to merge.

Returns

A single NumpyDataset containing all the samples from all datasets.

Return type

NumpyDataset

Example

```
>>> X1, y1 = np.random.rand(5, 3), np.random.randn(5, 1)
>>> first_dataset = dc.data.NumpyDataset(X1, y1)
>>> X2, y2 = np.random.rand(5, 3), np.random.randn(5, 1)
>>> second_dataset = dc.data.NumpyDataset(X2, y2)
>>> merged_dataset = dc.data.NumpyDataset.merge([first_dataset, second_dataset])
>>> print(len(merged_dataset) == len(first_dataset) + len(second_dataset))
True
```

static from_dataframe(*df: DataFrame, X: str | Sequence[str] | None = None, y: str | Sequence[str] | None = None, w: str | Sequence[str] | None = None, ids: str | None = None*)

Construct a Dataset from the contents of a pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – The pandas DataFrame
- **X** (*str or List[str], optional (default None)*) – The name of the column or columns containing the X array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **y** (*str or List[str], optional (default None)*) – The name of the column or columns containing the y array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **w** (*str or List[str], optional (default None)*) – The name of the column or columns containing the w array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **ids** (*str, optional (default None)*) – The name of the column containing the ids. If this is None, it will look for default column names that match those produced by to_dataframe().

get_statistics(*X_stats: bool = True, y_stats: bool = True*) → Tuple[ndarray, ...]

Compute and return statistics of this dataset.

Uses *self.itersamples()* to compute means and standard deviations of the dataset. Can compute on large datasets that don't fit in memory.

Parameters

- **X_stats** (*bool, optional (default True)*) – If True, compute feature-level mean and standard deviations.
- **y_stats** (*bool, optional (default True)*) – If True, compute label-level mean and standard deviations.

Returns

- If *X_stats == True*, returns *(X_means, X_stds)*.
- If *y_stats == True*, returns *(y_means, y_stds)*.
- If both are true, returns *(X_means, X_stds, y_means, y_stds)*.

Return type
Tuple

make_tf_dataset(*batch_size: int = 100*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*)

Create a tf.data.Dataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w) for one batch.

Parameters

- **batch_size** (*int, default 100*) – The number of samples to include in each batch.
- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset.
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **pad_batches** (*bool, default False*) – If True, batches are padded as necessary to make the size of each batch exactly equal batch_size.

Returns
TensorFlow Dataset that iterates over the same data.

Return type
tf.data.Dataset

Note: This class requires TensorFlow to be installed.

to_csv(*path: str*) → None

Write object to a comma-seperated values (CSV) file

Example

```
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> dataset.to_csv('out.csv')
```

Parameters
path (*str*) – File path or object

Return type
None

to_dataframe() → DataFrame

Construct a pandas DataFrame containing the data from this Dataset.

Returns
Pandas dataframe. If there is only a single feature per datapoint, will have column "X" else will have columns "X1,X2,..." for features. If there is only a single label per datapoint, will have column "y" else will have columns "y1,y2,..." for labels. If there is only a single weight per datapoint will have column "w" else will have columns "w1,w2,...". Will have column "ids" for identifiers.

Return type
pd.DataFrame

DiskDataset

The `dc.data.DiskDataset` class allows for the storage of larger datasets on disk. Each `DiskDataset` is associated with a directory in which it writes its contents to disk. Note that a `DiskDataset` can be very large, so some of the utility methods to access fields of a `Dataset` can be prohibitively expensive.

class DiskDataset(*data_dir: str*)

A Dataset that is stored as a set of files on disk.

The DiskDataset is the workhorse class of DeepChem that facilitates analyses on large datasets. Use this class whenever you're working with a large dataset that can't be easily manipulated in RAM.

On disk, a *DiskDataset* has a simple structure. All files for a given *DiskDataset* are stored in a *data_dir*. The contents of *data_dir* should be laid out as follows:

```
data_dir/
    |
    —> metadata.csv.gzip
    |
    —> tasks.json
    |
    —> shard-0-X.npy
    |
    —> shard-0-y.npy
    |
    —> shard-0-w.npy
    |
    —> shard-0-ids.npy
    |
    —> shard-1-X.npy
    .
    .
    .
```

The metadata is constructed by static method *DiskDataset._construct_metadata* and saved to disk by *DiskDataset._save_metadata*. The metadata itself consists of a csv file which has columns *('ids', 'X', 'y', 'w', 'ids_shape', 'X_shape', 'y_shape', 'w_shape')*. *tasks.json* consists of a list of task names for this dataset.

The actual data is stored in *.npy* files (numpy array files) of the form 'shard-0-X.npy', 'shard-0-y.npy', etc.

The basic structure of *DiskDataset* is quite robust and will likely serve you well for datasets up to about 100 GB or larger. However note that *DiskDataset* has not been tested for very large datasets at the terabyte range and beyond. You may be better served by implementing a custom *Dataset* class for those use cases.

Examples

Let's walk through a simple example of constructing a new *DiskDataset*.

```
>>> import deepchem as dc
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
```

If you have already saved a *DiskDataset* to *data_dir*, you can reinitialize it with

>> data_dir = "/path/to/my/data" >> dataset = dc.data.DiskDataset(data_dir)

Once you have a dataset you can access its attributes as follows

```
>>> X = np.random.rand(10, 10)
>>> y = np.random.rand(10,)
>>> w = np.ones_like(y)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> X, y, w = dataset.X, dataset.y, dataset.w
```

One thing to beware of is that *dataset.X*, *dataset.y*, *dataset.w* are loading data from disk! If you have a large dataset, these operations can be extremely slow. Instead try iterating through the dataset instead.

```
>>> for (xi, yi, wi, idi) in dataset.itersamples():
...   pass
```

`data_dir`

Location of directory where this *DiskDataset* is stored to disk

Type
str

`metadata_df`

Pandas Dataframe holding metadata for this *DiskDataset*

Type
pd.DataFrame

`legacy_metadata`

Whether this *DiskDataset* uses legacy format.

Type
bool

Note: *DiskDataset* originally had a simpler metadata format without shape information. Older *DiskDataset* objects had metadata files with columns *('ids', 'X', 'y', 'w')* and not additional shape columns. *DiskDataset* maintains backwards compatibility with this older metadata format, but we recommend for performance reasons not using legacy metadata for new projects.

`__init__`(*data_dir: str*) → None

Load a constructed DiskDataset from disk

Note that this method cannot construct a new disk dataset. Instead use static methods *DiskDataset.create_dataset* or *DiskDataset.from_numpy* for that purpose. Use this constructor instead to load a *DiskDataset* that has already been created on disk.

Parameters

`data_dir` (*`str`*) – Location on disk of an existing *DiskDataset*.

`static` **`create_dataset`**(*shard_generator: Iterable[Tuple[ndarray, ndarray, ndarray, ndarray]]*, *data_dir: str | None = None*, *tasks: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*) → *DiskDataset*

Creates a new DiskDataset

Parameters

- **`shard_generator`** (*`Iterable[Batch]`*) – An iterable (either a list or generator) that provides tuples of data (X, y, w, ids). Each tuple will be written to a separate shard on disk.
- **`data_dir`** (*`str, optional (default None)`*) – Filename for data directory. Creates a temp directory if none specified.
- **`tasks`** (*`Sequence, optional (default [])`*) – List of tasks for this dataset.

Returns

A new *DiskDataset* constructed from the given data

Return type

DiskDataset

`load_metadata`() → Tuple[List[str], DataFrame]

Helper method that loads metadata from disk.

`static` **`write_data_to_disk`**(*data_dir: str*, *basename: str*, *X: ndarray | None = None*, *y: ndarray | None = None*, *w: ndarray | None = None*, *ids: ndarray | None = None*) → List[Any]

Static helper method to write data to disk.

This helper method is used to write a shard of data to disk.

Parameters

- **`data_dir`** (*`str`*) – Data directory to write shard to.
- **`basename`** (*`str`*) – Basename for the shard in question.
- **`X`** (*`np.ndarray, optional (default None)`*) – The features array.
- **`y`** (*`np.ndarray, optional (default None)`*) – The labels array.
- **`w`** (*`np.ndarray, optional (default None)`*) – The weights array.
- **`ids`** (*`np.ndarray, optional (default None)`*) – The identifiers array.

Returns

List with values *[out_ids, out_X, out_y, out_w, out_ids_shape, out_X_shape, out_y_shape, out_w_shape]* with filenames of locations to disk which these respective arrays were written.

Return type

List[Optional[str]]

save_to_disk() → None

Save dataset to disk.

move(*new_data_dir: str*, *delete_if_exists: bool | None = True*) → None

Moves dataset to new directory.

Parameters

- **new_data_dir** (*str*) – The new directory name to move this to dataset to.
- **delete_if_exists** (*bool, optional (default True)*) – If this option is set, delete the destination directory if it exists before moving. This is set to True by default to be backwards compatible with behavior in earlier versions of DeepChem.

Note: This is a stateful operation! *self.data_dir* will be moved into *new_data_dir*. If *delete_if_exists* is set to *True* (by default this is set *True*), then *new_data_dir* is deleted if it's a pre-existing directory.

copy(*new_data_dir: str*) → *DiskDataset*

Copies dataset to new directory.

Parameters

new_data_dir (*str*) – The new directory name to copy this to dataset to.

Returns

A copied DiskDataset object.

Return type

DiskDataset

Note: This is a stateful operation! Any data at *new_data_dir* will be deleted and *self.data_dir* will be deep copied into *new_data_dir*.

get_task_names() → ndarray

Gets learning tasks associated with this dataset.

reshard(*shard_size: int*) → None

Reshards data to have specified shard size.

Parameters

shard_size (*int*) – The size of shard.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> X = np.random.rand(100, 10)
>>> d = dc.data.DiskDataset.from_numpy(X)
>>> d.reshard(shard_size=10)
>>> d.get_number_shards()
10
```

Note: If this *DiskDataset* is in *legacy_metadata* format, reshard will convert this dataset to have non-legacy metadata.

get_data_shape() → Tuple[int, ...]

Gets array shape of datapoints in this dataset.

get_shard_size() → int

Gets size of shards on disk.

get_number_shards() → int

Returns the number of shards for this dataset.

itershards() → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Return an object that iterates over all shards in dataset.

Datasets are stored in sharded fashion on disk. Each call to next() for the generator defined by this function returns the data from a particular shard. The order of shards returned is guaranteed to remain fixed.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

iterbatches(*batch_size: int | None = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over minibatches from the dataset.

It is guaranteed that the number of batches returned is *math.ceil(len(dataset)/batch_size)*. Each minibatch is returned as a tuple of four numpy arrays: *(X, y, w, ids)*.

Parameters

- **batch_size** (`int, optional (default None)`) – Number of elements in a batch. If None, then it yields batches with size equal to the size of each individual shard.
- **epoch** (`int, default 1`) – Number of epochs to walk over dataset
- **deterministic** (`bool, default False`) – Whether or not we should should shuffle each shard before generating the batches. Note that this is only local in the sense that it does not ever mix between different shards.
- **pad_batches** (`bool, default False`) – Whether or not we should pad the last batch, globally, such that it has exactly batch_size elements.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

itersamples() → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over the samples in the dataset.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

Examples

```
>>> dataset = DiskDataset.from_numpy(np.ones((2,2)), np.ones((2,1)))
>>> for x, y, w, id in dataset.itersamples():
...   print(x.tolist(), y.tolist(), w.tolist(), id)
[1.0, 1.0] [1.0] [1.0] 0
[1.0, 1.0] [1.0] [1.0] 1
```

transform(*transformer: Transformer*, *parallel: bool = False*, *out_dir: str | None = None*, ***args*) → *DiskDataset*

Construct a new dataset by applying a transformation to every sample in this dataset.

The argument is a function that can be called as follows: >> newx, newy, neww = fn(x, y, w)

It might be called only once with the whole dataset, or multiple times with different subsets of the data. Each time it is called, it should transform the samples and return the transformed data.

Parameters

- **transformer** (*dc.trans.Transformer*) – The transformation to apply to each sample in the dataset.
- **parallel** (*bool, default False*) – If True, use multiple processes to transform the dataset in parallel.
- **out_dir** (*str, optional (default None)*) – The directory to save the new dataset in. If this is omitted, a temporary directory is created automaticall.

Returns
A newly constructed Dataset object

Return type
DiskDataset

make_pytorch_dataset(*epochs: int = 1*, *deterministic: bool = False*, *batch_size: int | None = None*)

Create a torch.utils.data.IterableDataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w, id) containing the data for one batch, or for a single sample if batch_size is None.

Parameters

- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **batch_size** (*int, optional (default None)*) – The number of samples to return in each batch. If None, each returned value is a single sample.

Returns
torch.utils.data.IterableDataset that iterates over the data in this dataset.

Return type
torch.utils.data.IterableDataset

Note: This method requires PyTorch to be installed.

static from_numpy(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ids: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *tasks: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *data_dir: str | None = None*) → *DiskDataset*

Creates a DiskDataset object from specified Numpy arrays.

Parameters

- **X** (*np.ndarray*) – Feature array.
- **y** (*np.ndarray, optional (default None)*) – Labels array.
- **w** (*np.ndarray, optional (default None)*) – Weights array.
- **ids** (*np.ndarray, optional (default None)*) – Identifiers array.
- **tasks** (*Sequence, optional (default None)*) – Tasks in this dataset
- **data_dir** (*str, optional (default None)*) – The directory to write this dataset to. If none is specified, will use a temporary directory instead.

Returns
A new *DiskDataset* constructed from the provided information.

Return type
DiskDataset

static merge(*datasets: Iterable[Dataset]*, *merge_dir: str | None = None*) → *DiskDataset*

Merges provided datasets into a merged dataset.

Parameters

- **datasets** (*Iterable[Dataset]*) – List of datasets to merge.
- **merge_dir** (*str, optional (default None)*) – The new directory path to store the merged DiskDataset.

Returns
A merged DiskDataset.

Return type
DiskDataset

subset(*shard_nums: Sequence[int]*, *subset_dir: str | None = None*) → *DiskDataset*

Creates a subset of the original dataset on disk.

Parameters

- **shard_nums** (*Sequence[int]*) – The indices of shard to extract from the original DiskDataset.
- **subset_dir** (*str, optional (default None)*) – The new directory path to store the subset DiskDataset.

Returns
A subset DiskDataset.

Return type
DiskDataset

`sparse_shuffle()` → None

Shuffling that exploits data sparsity to shuffle large datasets.

If feature vectors are sparse, say circular fingerprints or any other representation that contains few nonzero values, it can be possible to exploit the sparsity of the vector to simplify shuffles. This method implements a sparse shuffle by compressing sparse feature vectors down into a compressed representation, then shuffles this compressed dataset in memory and writes the results to disk.

Note: This method only works for 1-dimensional feature vectors (does not work for tensorial featurizations). Note that this shuffle is performed in place.

`complete_shuffle`(*data_dir: str | None = None*) → *Dataset*

Completely shuffle across all data, across all shards.

Note: The algorithm used for this complete shuffle is O(N^2) where N is the number of shards. It simply constructs each shard of the output dataset one at a time. Since the complete shuffle can take a long time, it's useful to watch the logging output. Each shuffled shard is constructed using select() which logs as it selects from each original shard. This will results in O(N^2) logging statements, one for each extraction of shuffled shard i's contributions from original shard j.

Parameters
data_dir (*Optional[str], (default None)*) – Directory to write the shuffled dataset to. If none is specified a temporary directory will be used.

Returns
A DiskDataset whose data is a randomly shuffled version of this dataset.

Return type
DiskDataset

`shuffle_each_shard`(*shard_basenames: List[str] | None = None*) → None

Shuffles elements within each shard of the dataset.

Parameters
shard_basenames (*List[str], optional (default None)*) – The basenames for each shard. If this isn't specified, will assume the basenames of form "shard-i" used by *create_dataset* and *reshard*.

`shuffle_shards()` → None

Shuffles the order of the shards for this dataset.

`get_shard`(*i: int*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Retrieves data for the i-th shard from disk.

Parameters
i (*int*) – Shard index for shard to retrieve batch from.

Returns
A batch data for i-th shard.

Return type
Batch

`get_shard_ids`(*i: int*) → ndarray

Retrieves the list of IDs for the i-th shard from disk.

Parameters
i (*int*) – Shard index for shard to retrieve weights from.

Returns
A numpy array of ids for i-th shard.

Return type
np.ndarray

`get_shard_y`(*i: int*) → ndarray

Retrieves the labels for the i-th shard from disk.

Parameters
i (*int*) – Shard index for shard to retrieve labels from.

Returns
A numpy array of labels for i-th shard.

Return type
np.ndarray

`get_shard_w`(*i: int*) → ndarray

Retrieves the weights for the i-th shard from disk.

Parameters
i (*int*) – Shard index for shard to retrieve weights from.

Returns
A numpy array of weights for i-th shard.

Return type
np.ndarray

`add_shard`(*X: ndarray*, *y: ndarray | None = None*, *w: ndarray | None = None*, *ids: ndarray | None = None*) → None

Adds a data shard.

Parameters

- **X** (*np.ndarray*) – Feature array.
- **y** (*np.ndarray, optioanl (default None)*) – Labels array.
- **w** (*np.ndarray, optioanl (default None)*) – Weights array.
- **ids** (*np.ndarray, optioanl (default None)*) – Identifiers array.

`set_shard`(*shard_num: int*, *X: ndarray*, *y: ndarray | None = None*, *w: ndarray | None = None*, *ids: ndarray | None = None*) → None

Writes data shard to disk.

Parameters

- **shard_num** (*int*) – Shard index for shard to set new data.
- **X** (*np.ndarray*) – Feature array.
- **y** (*np.ndarray, optioanl (default None)*) – Labels array.

- **w** (*np.ndarray, optioanl (default None)*) – Weights array.
- **ids** (*np.ndarray, optioanl (default None)*) – Identifiers array.

select(*indices: Sequence[int] | ndarray*, *select_dir: str | None = None*, *select_shard_size: int | None = None*, *output_numpy_dataset: bool | None = False*) → *Dataset*

Creates a new dataset from a selection of indices from self.

Examples

```
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> selected = dataset.select([1, 3, 4])
>>> len(selected)
3
```

Parameters

- **indices** (*Sequence*) – List of indices to select.
- **select_dir** (*str, optional (default None)*) – Path to new directory that the selected indices will be copied to.
- **select_shard_size** (*Optional[int], (default None)*) – If specified, the shard-size to use for output selected *DiskDataset*. If not output_numpy_dataset, then this is set to this current dataset's shard size if not manually specified.
- **output_numpy_dataset** (*Optional[bool], (default False)*) – If True, output an in-memory *NumpyDataset* instead of a *DiskDataset*. Note that *select_dir* and *select_shard_size* must be *None* if this is *True*

Returns

A dataset containing the selected samples. The default dataset is *DiskDataset*. If *output_numpy_dataset* is True, the dataset is *NumpyDataset*.

Return type

Dataset

property ids: ndarray

Get the ids vector for this dataset as a single numpy array.

property X: ndarray

Get the X vector for this dataset as a single numpy array.

property y: ndarray

Get the y vector for this dataset as a single numpy array.

property w: ndarray

Get the weight vector for this dataset as a single numpy array.

property memory_cache_size: int

Get the size of the memory cache for this dataset, measured in bytes.

__len__() → int

Finds number of elements in dataset.

`get_shape()` → Tuple[Tuple[int, ...], Tuple[int, ...], Tuple[int, ...], Tuple[int, ...]]

Finds shape of dataset.

Returns four tuples, giving the shape of the X, y, w, and ids arrays.

`get_label_means()` → DataFrame

Return pandas series of label means.

`get_label_stds()` → DataFrame

Return pandas series of label stds.

`static from_dataframe`(*df: DataFrame*, *X: str | Sequence[str] | None = None*, *y: str | Sequence[str] | None = None*, *w: str | Sequence[str] | None = None*, *ids: str | None = None*)

Construct a Dataset from the contents of a pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – The pandas DataFrame
- **X** (*str or List[str], optional (default None)*) – The name of the column or columns containing the X array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **y** (*str or List[str], optional (default None)*) – The name of the column or columns containing the y array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **w** (*str or List[str], optional (default None)*) – The name of the column or columns containing the w array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **ids** (*str, optional (default None)*) – The name of the column containing the ids. If this is None, it will look for default column names that match those produced by to_dataframe().

`get_statistics`(*X_stats: bool = True*, *y_stats: bool = True*) → Tuple[ndarray, ...]

Compute and return statistics of this dataset.

Uses *self.itersamples()* to compute means and standard deviations of the dataset. Can compute on large datasets that don't fit in memory.

Parameters

- **X_stats** (*bool, optional (default True)*) – If True, compute feature-level mean and standard deviations.
- **y_stats** (*bool, optional (default True)*) – If True, compute label-level mean and standard deviations.

Returns

- If *X_stats == True*, returns *(X_means, X_stds)*.
- If *y_stats == True*, returns *(y_means, y_stds)*.
- If both are true, returns *(X_means, X_stds, y_means, y_stds)*.

Return type

Tuple

make_tf_dataset(*batch_size: int = 100*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*)

Create a tf.data.Dataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w) for one batch.

Parameters

- **batch_size** (*int, default 100*) – The number of samples to include in each batch.
- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset.
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **pad_batches** (*bool, default False*) – If True, batches are padded as necessary to make the size of each batch exactly equal batch_size.

Returns

TensorFlow Dataset that iterates over the same data.

Return type

tf.data.Dataset

Note: This class requires TensorFlow to be installed.

to_csv(*path: str*) → None

Write object to a comma-seperated values (CSV) file

Example

```
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> dataset.to_csv('out.csv')
```

Parameters

path (*str*) – File path or object

Return type

None

to_dataframe() → DataFrame

Construct a pandas DataFrame containing the data from this Dataset.

Returns

Pandas dataframe. If there is only a single feature per datapoint, will have column "X" else will have columns "X1,X2,..." for features. If there is only a single label per datapoint, will have column "y" else will have columns "y1,y2,..." for labels. If there is only a single weight per datapoint will have column "w" else will have columns "w1,w2,...". Will have column "ids" for identifiers.

Return type

pd.DataFrame

ImageDataset

The `dc.data.ImageDataset` class is optimized to allow for convenient processing of image based datasets.

class ImageDataset(*X: ndarray | List[str]*, *y: ndarray | List[str] | None*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ids: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*)

A Dataset that loads data from image files on disk.

__init__(*X: ndarray | List[str]*, *y: ndarray | List[str] | None*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *ids: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*) → None

Create a dataset whose X and/or y array is defined by image files on disk.

Parameters

- **X** (*np.ndarray or List[str]*) – The dataset's input data. This may be either a single NumPy array directly containing the data, or a list containing the paths to the image files
- **y** (*np.ndarray or List[str]*) – The dataset's labels. This may be either a single NumPy array directly containing the data, or a list containing the paths to the image files
- **w** (*np.ndarray, optional (default None)*) – a 1D or 2D array containing the weights for each sample or sample/task pair
- **ids** (*np.ndarray, optional (default None)*) – the sample IDs

__len__() → int

Get the number of elements in the dataset.

get_shape() → Tuple[Tuple[int, ...], Tuple[int, ...], Tuple[int, ...], Tuple[int, ...]]

Get the shape of the dataset.

Returns four tuples, giving the shape of the X, y, w, and ids arrays.

get_task_names() → ndarray

Get the names of the tasks associated with this dataset.

property X: ndarray

Get the X vector for this dataset as a single numpy array.

property y: ndarray

Get the y vector for this dataset as a single numpy array.

property ids: ndarray

Get the ids vector for this dataset as a single numpy array.

property w: ndarray

Get the weight vector for this dataset as a single numpy array.

iterbatches(*batch_size: int | None = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over minibatches from the dataset.

Each minibatch is returned as a tuple of four numpy arrays: *(X, y, w, ids)*.

Parameters

- **batch_size** (*int, optional (default None)*) – Number of elements in each batch.
- **epochs** (*int, default 1*) – Number of epochs to walk over dataset.
- **deterministic** (*bool, default False*) – If True, follow deterministic order.
- **pad_batches** (*bool, default False*) – If True, pad each batch to *batch_size*.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

itersamples() → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over the samples in the dataset.

Returns
Iterator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

transform(*transformer:* Transformer, ***args*) → *NumpyDataset*

Construct a new dataset by applying a transformation to every sample in this dataset.

The argument is a function that can be called as follows:

>> newx, newy, neww = fn(x, y, w)

It might be called only once with the whole dataset, or multiple times with different subsets of the data. Each time it is called, it should transform the samples and return the transformed data.

Parameters
transformer (*dc.trans.Transformer*) – The transformation to apply to each sample in the dataset

Returns
A newly constructed NumpyDataset object

Return type
NumpyDataset

select(*indices: Sequence[int] | ndarray*, *select_dir: str | None = None*) → *ImageDataset*

Creates a new dataset from a selection of indices from self.

Parameters

- **indices** (*Sequence*) – List of indices to select.
- **select_dir** (*str, optional (default None)*) – Used to provide same API as *DiskDataset*. Ignored since *ImageDataset* is purely in-memory.

Returns
A selected ImageDataset object

Return type
ImageDataset

`make_pytorch_dataset`(*epochs: int = 1*, *deterministic: bool = False*, *batch_size: int | None = None*)

Create a torch.utils.data.IterableDataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w, id) containing the data for one batch, or for a single sample if batch_size is None.

Parameters

- `epochs` (*int, default 1*) – The number of times to iterate over the Dataset.
- `deterministic` (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- `batch_size` (*int, optional (default None)*) – The number of samples to return in each batch. If None, each returned value is a single sample.

Returns
torch.utils.data.IterableDataset that iterates over the data in this dataset.

Return type
torch.utils.data.IterableDataset

Note: This method requires PyTorch to be installed.

`static from_dataframe`(*df: DataFrame*, *X: str | Sequence[str] | None = None*, *y: str | Sequence[str] | None = None*, *w: str | Sequence[str] | None = None*, *ids: str | None = None*)

Construct a Dataset from the contents of a pandas DataFrame.

Parameters

- `df` (*pd.DataFrame*) – The pandas DataFrame
- `X` (*str or List[str], optional (default None)*) – The name of the column or columns containing the X array. If this is None, it will look for default column names that match those produced by to_dataframe().
- `y` (*str or List[str], optional (default None)*) – The name of the column or columns containing the y array. If this is None, it will look for default column names that match those produced by to_dataframe().
- `w` (*str or List[str], optional (default None)*) – The name of the column or columns containing the w array. If this is None, it will look for default column names that match those produced by to_dataframe().
- `ids` (*str, optional (default None)*) – The name of the column containing the ids. If this is None, it will look for default column names that match those produced by to_dataframe().

`get_statistics`(*X_stats: bool = True*, *y_stats: bool = True*) → Tuple[ndarray, ...]

Compute and return statistics of this dataset.

Uses *self.itersamples()* to compute means and standard deviations of the dataset. Can compute on large datasets that don't fit in memory.

Parameters

- `X_stats` (*bool, optional (default True)*) – If True, compute feature-level mean and standard deviations.
- `y_stats` (*bool, optional (default True)*) – If True, compute label-level mean and standard deviations.

Returns

- If *X_stats == True*, returns *(X_means, X_stds)*.
- If *y_stats == True*, returns *(y_means, y_stds)*.
- If both are true, returns *(X_means, X_stds, y_means, y_stds)*.

Return type
Tuple

make_tf_dataset(*batch_size: int = 100*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*)

Create a tf.data.Dataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w) for one batch.

Parameters

- **batch_size** (*int, default 100*) – The number of samples to include in each batch.
- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset.
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **pad_batches** (*bool, default False*) – If True, batches are padded as necessary to make the size of each batch exactly equal batch_size.

Returns
TensorFlow Dataset that iterates over the same data.

Return type
tf.data.Dataset

Note: This class requires TensorFlow to be installed.

to_csv(*path: str*) → None

Write object to a comma-seperated values (CSV) file

Example

```
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> dataset.to_csv('out.csv')
```

Parameters
path (*str*) – File path or object

Return type
None

to_dataframe() → DataFrame

Construct a pandas DataFrame containing the data from this Dataset.

Returns
Pandas dataframe. If there is only a single feature per datapoint, will have column "X" else will have columns "X1,X2,…" for features. If there is only a single label per datapoint, will have column "y" else will have columns "y1,y2,…" for labels. If there is only a single weight per datapoint will have column "w" else will have columns "w1,w2,…". Will have column "ids" for identifiers.

Return type
pd.DataFrame

3.8.2 Data Loaders

Processing large amounts of input data to construct a `dc.data.Dataset` object can require some amount of hacking. To simplify this process for you, you can use the `dc.data.DataLoader` classes. These classes provide utilities for you to load and process large amounts of data.

CSVLoader

class CSVLoader(*tasks: List[str]*, *featurizer: Featurizer*, *feature_field: str | None = None*, *id_field: str | None = None*, *smiles_field: str | None = None*, *log_every_n: int = 1000*)

Creates *Dataset* objects from input CSV files.

This class provides conveniences to load data from CSV files. It's possible to directly featurize data from CSV files using pandas, but this class may prove useful if you're processing large CSV files that you don't want to manipulate directly in memory. Note that samples which cannot be featurized are filtered out in the creation of final dataset.

Examples

Let's suppose we have some smiles and labels

```
>>> smiles = ["C", "CCC"]
>>> labels = [1.5, 2.3]
```

Let's put these in a dataframe.

```
>>> import pandas as pd
>>> df = pd.DataFrame(list(zip(smiles, labels)), columns=["smiles", "task1"])
```

Let's now write this to disk somewhere. We can now use *CSVLoader* to process this CSV dataset.

```
>>> import tempfile
>>> import deepchem as dc
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_csv(tmpfile.name)
...     loader = dc.data.CSVLoader(["task1"], feature_field="smiles",
...                                featurizer=dc.feat.CircularFingerprint())
...     dataset = loader.create_dataset(tmpfile.name)
>>> len(dataset)
2
```

Of course in practice you should already have your data in a CSV file if you're using *CSVLoader*. If your data is already in memory, use *InMemoryLoader* instead.

Sometimes there will be datasets without specific tasks, for example datasets which are used in unsupervised learning tasks. Such datasets can be loaded by leaving the *tasks* field empty.

Example

```
>>> x1, x2 = [2, 3, 4], [4, 6, 8]
>>> df = pd.DataFrame({"x1":x1, "x2": x2}).reset_index()
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_csv(tmpfile.name)
...     loader = dc.data.CSVLoader(tasks=[], id_field="index", feature_field=["x1",
↪"x2"],
...                                featurizer=dc.feat.DummyFeaturizer())
...     dataset = loader.create_dataset(tmpfile.name)
>>> len(dataset)
3
```

`__init__`(*tasks: List[str]*, *featurizer: Featurizer*, *feature_field: str | None = None*, *id_field: str | None = None*, *smiles_field: str | None = None*, *log_every_n: int = 1000*)

Initializes CSVLoader.

Parameters

- **`tasks`** (*`List[str]`*) – List of task names
- **`featurizer`** (`Featurizer`) – Featurizer to use to process data.
- **`feature_field`** (*`str, optional (default None)`*) – Field with data to be featurized.
- **`id_field`** (*`str, optional, (default None)`*) – CSV column that holds sample identifier
- **`smiles_field`** (*`str, optional (default None) (DEPRECATED)`*) – Name of field that holds smiles string.
- **`log_every_n`** (*`int, optional (default 1000)`*) – Writes a logging statement this often.

`create_dataset`(*inputs: Any | Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided files.

Reads in *inputs* and uses *self.featurizer* to featurize the data in these inputs. For large files, automatically shards into smaller chunks of *shard_size* datapoints for convenience. Returns a *Dataset* object that contains the featurized dataset.

This implementation assumes that the helper methods *_get_shards* and *_featurize_shard* are implemented and that each shard returned by *_get_shards* is a pandas dataframe. You may choose to reuse or override this method in your subclass implementations.

Parameters

- **`inputs`** (*`List`*) – List of inputs to process. Entries can be filenames or arbitrary objects.
- **`data_dir`** (*`str, optional (default None)`*) – Directory to store featurized dataset.
- **`shard_size`** (*`int, optional (default 8192)`*) – Number of examples stored in each shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type
DiskDataset

UserCSVLoader

`class UserCSVLoader`(*tasks: List[str]*, *featurizer:* Featurizer, *feature_field: str | None = None*, *id_field: str | None = None*, *smiles_field: str | None = None*, *log_every_n: int = 1000*)

Handles loading of CSV files with user-defined features.

This is a convenience class that allows for descriptors already present in a CSV file to be extracted without any featurization necessary.

Examples

Let's suppose we have some descriptors and labels. (Imagine that these descriptors have been computed by an external program.)

```
>>> desc1 = [1, 43]
>>> desc2 = [-2, -22]
>>> labels = [1.5, 2.3]
>>> ids = ["cp1", "cp2"]
```

Let's put these in a dataframe.

```
>>> import pandas as pd
>>> df = pd.DataFrame(list(zip(ids, desc1, desc2, labels)), columns=["id", "desc1",
↪"desc2", "task1"])
```

Let's now write this to disk somewhere. We can now use *UserCSVLoader* to process this CSV dataset.

```
>>> import tempfile
>>> import deepchem as dc
>>> featurizer = dc.feat.UserDefinedFeaturizer(["desc1", "desc2"])
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_csv(tmpfile.name)
...     loader = dc.data.UserCSVLoader(["task1"], id_field="id",
...                          featurizer=featurizer)
...     dataset = loader.create_dataset(tmpfile.name)
>>> len(dataset)
2
>>> dataset.X[0, 0]
1
```

The difference between *UserCSVLoader* and *CSVLoader* is that our descriptors (our features) have already been computed for us, but are spread across multiple columns of the CSV file.

Of course in practice you should already have your data in a CSV file if you're using *UserCSVLoader*. If your data is already in memory, use *InMemoryLoader* instead.

`__init__`(*tasks: List[str]*, *featurizer:* Featurizer, *feature_field: str | None = None*, *id_field: str | None = None*, *smiles_field: str | None = None*, *log_every_n: int = 1000*)

Initializes CSVLoader.

Parameters

- **tasks** (*List[str]*) – List of task names
- **featurizer** (Featurizer) – Featurizer to use to process data.
- **feature_field** (*str, optional (default None)*) – Field with data to be featurized.
- **id_field** (*str, optional, (default None)*) – CSV column that holds sample identifier
- **smiles_field** (*str, optional (default None) (DEPRECATED)*) – Name of field that holds smiles string.
- **log_every_n** (*int, optional (default 1000)*) – Writes a logging statement this often.

create_dataset(*inputs: Any | Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided files.

Reads in *inputs* and uses *self.featurizer* to featurize the data in these inputs. For large files, automatically shards into smaller chunks of *shard_size* datapoints for convenience. Returns a *Dataset* object that contains the featurized dataset.

This implementation assumes that the helper methods *_get_shards* and *_featurize_shard* are implemented and that each shard returned by *_get_shards* is a pandas dataframe. You may choose to reuse or override this method in your subclass implementations.

Parameters

- **inputs** (*List*) – List of inputs to process. Entries can be filenames or arbitrary objects.
- **data_dir** (*str, optional (default None)*) – Directory to store featurized dataset.
- **shard_size** (*int, optional (default 8192)*) – Number of examples stored in each shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type

DiskDataset

ImageLoader

class ImageLoader(*tasks: List[str] | None = None*, *sorting: bool = True*)

Creates *Dataset* objects from input image files.

This class allows for loading of images in various formats. For user convenience, also accepts zip-files and directories of images and uses some limited intelligence to attempt to traverse subdirectories which contain images.

Currently, only .png and .tif files are supported. If the inputs or labels are given as a list of files, the list must contain only image files.

Examples

For this example, we will be using the BBBC001 Dataset. This dataset contains 6 images of human HT29 colon cancer cells. We will use the images as inputs and we will assign the labels as integers ranging from 1 to 6 for the sake of simplicity.

To learn more about this dataset, please visit: https://data.broadinstitute.org/bbbc/BBBC001/ and also see our loader for this dataset: *deepchem.molnet.loadbbbc001*.

Let's begin by importing the necessary modules and downloading the dataset. >>> import os >>> import deepchem as dc >>> data_dir = dc.utils.data_utils.get_data_dir() >>> dataset_file = os.path.join(data_dir, "BBBC001_v1_images_tif.zip") >>> BBBC1_IMAGE_URL = 'https://data.broadinstitute.org/bbbc/BBBC001/BBBC001_v1_images_tif.zip' >>> if not os.path.exists(dataset_file): ... dc.utils.data_utils.download_url(url=BBBC1_IMAGE_URL, dest_dir=data_dir)

Now that we have the dataset, let's create a list of labels for each image.

```
>>> labels = np.array([1,2,3,4,5,6])
```

Let's now write this to disk somewhere. We can now use *ImageLoader* to process this Image dataset. We do not use a featurizer here, hence the *UserDefinedFeaturizer* with an empty list.

```
>>> featurizer = dc.feat.UserDefinedFeaturizer([])
>>> loader = dc.data.ImageLoader(tasks=['demo-task'], sorting=False)
>>> dataset = loader.create_dataset(inputs=(dataset_file, labels),
...                                 in_memory=False)
```

We can confirm that we have 6 images in our dataset and 6 labels. The images are of size 512x512 while the labels are just integers.

```
>>> len(dataset)
6
>>> dataset.X.shape
(6, 512, 512)
>>> dataset.y.shape
(6,)
```

The label files can also be images similar to the inputs, in which case we can provide a list of label files instead of a list of labels.

To show this, we will use the input data as the ground truths, this is often seen when making autoencoders. Similar to the above example, let's use *ImageLoader* to process this Image dataset.

```
>>> featurizer = dc.feat.UserDefinedFeaturizer([])
>>> loader = dc.data.ImageLoader(tasks=['demo-task'], sorting=False)
>>> dataset = loader.create_dataset(inputs=(dataset_file, dataset_file),
...                                 in_memory=False)
```

We can confirm that we have 6 images in our dataset and 6 labels. The images are of size 512x512 while the labels are also images of size 512x512.

```
>>> len(dataset)
6
>>> dataset.X.shape
(6, 512, 512)
>>> dataset.y.shape
(6, 512, 512)
```

`__init__`(*tasks: List[str] | None = None*, *sorting: bool = True*)

Initialize image loader.

At present, custom image featurizers aren't supported by this loader class.

Parameters

- `tasks` (`List[str], optional (default None)`) – List of task names for image labels.
- `sorting` (`bool, optional (default True)`) – Whether to sort image files by filename.

`create_dataset`(*inputs: str | Sequence[str] | Tuple[Any] | Tuple[str, Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*, *in_memory: bool = False*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided image files and labels/weights.

Parameters

- `inputs` (*Union[OneOrMany[str], Tuple[Any]]*) – The inputs provided should be one of the following
  - filename
  - list of filenames
  - Tuple (list of filenames, labels)
  - Tuple (list of filenames, list of label filenames)
  - Tuple (list of filenames, labels, weights)
  - Tuple (list of filenames, list of label filenames, weights)

  Each file in a given list of filenames should either be of a supported image format (.png, .tif only for now) or of a compressed folder of image files (only .zip for now). If *labels* or *weights* are provided, they must correspond to the sorted order of all filenames provided, with one label/weight per file. Labels can be filenames too, in which case the labels are loaded as images.
- `data_dir` (`str, optional (default None)`) – Directory to store featurized dataset.
- `shard_size` (`int, optional (default 8192)`) – Shard size when loading data.
- `in_memory` (`bool, optioanl (default False)`) – If true, return in-memory *NumpyDataset*. Else return *ImageDataset*.

Returns

- if *in_memory == False*, the return value is *ImageDataset*.
- if *in_memory == True* and *data_dir is None*, the return value is *NumpyDataset*.
- if *in_memory == True* and *data_dir is not None*, the return value is *DiskDataset*.

Return type

ImageDataset or *NumpyDataset* or *DiskDataset*

JsonLoader

JSON is a flexible file format that is human-readable, lightweight, and more compact than other open standard formats like XML. JSON files are similar to python dictionaries of key-value pairs. All keys must be strings, but values can be any of (string, number, object, array, boolean, or null), so the format is more flexible than CSV. JSON is used for describing structured data and to serialize objects. It is conveniently used to read/write Pandas dataframes with the *pandas.read_json* and *pandas.write_json* methods.

class JsonLoader(*tasks: List[str]*, *feature_field: str*, *featurizer: Featurizer*, *label_field: str | None = None*, *weight_field: str | None = None*, *id_field: str | None = None*, *log_every_n: int = 1000*)

Creates *Dataset* objects from input json files.

This class provides conveniences to load data from json files. It's possible to directly featurize data from json files using pandas, but this class may prove useful if you're processing large json files that you don't want to manipulate directly in memory.

It is meant to load JSON files formatted as "records" in line delimited format, which allows for sharding. `list like [{column -> value}, ... , {column -> value}]`.

Examples

Let's create the sample dataframe.

```
>>> composition = ["LiCoO2", "MnO2"]
>>> labels = [1.5, 2.3]
>>> import pandas as pd
>>> df = pd.DataFrame(list(zip(composition, labels)), columns=["composition", "task
↪"])
```

Dump the dataframe to the JSON file formatted as "records" in line delimited format and load the json file by JsonLoader.

```
>>> import tempfile
>>> import deepchem as dc
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_json(tmpfile.name, orient='records', lines=True)
...     featurizer = dc.feat.ElementPropertyFingerprint()
...     loader = dc.data.JsonLoader(["task"], feature_field="composition",␣
↪featurizer=featurizer)
...     dataset = loader.create_dataset(tmpfile.name)
>>> len(dataset)
2
```

__init__(*tasks: List[str]*, *feature_field: str*, *featurizer: Featurizer*, *label_field: str | None = None*, *weight_field: str | None = None*, *id_field: str | None = None*, *log_every_n: int = 1000*)

Initializes JsonLoader.

Parameters

- **tasks** (`List[str]`) – List of task names
- **feature_field** (`str`) – JSON field with data to be featurized.
- **featurizer** (`Featurizer`) – Featurizer to use to process data
- **label_field** (`str, optional (default None)`) – Field with target variables.
- **weight_field** (`str, optional (default None)`) – Field with weights.

- **id_field** (*str, optional (default None)*) – Field for identifying samples.
- **log_every_n** (*int, optional (default 1000)*) – Writes a logging statement this often.

create_dataset(*input_files: str | Sequence[str]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *DiskDataset*

Creates a *Dataset* from input JSON files.

Parameters

- **input_files** (*OneOrMany[str]*) – List of JSON filenames.
- **data_dir** (*Optional[str], default None*) – Name of directory where featurized data is stored.
- **shard_size** (*int, optional (default 8192)*) – Shard size when loading data.

Returns

A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type

DiskDataset

SDFLoader

class SDFLoader(*tasks: List[str]*, *featurizer: Featurizer*, *sanitize: bool = False*, *log_every_n: int = 1000*)

Creates a *Dataset* object from SDF input files.

This class provides conveniences to load and featurize data from Structure Data Files (SDFs). SDF is a standard format for structural information (3D coordinates of atoms and bonds) of molecular compounds.

Examples

```
>>> import deepchem as dc
>>> import os
>>> current_dir = os.path.dirname(os.path.realpath(__file__))
>>> featurizer = dc.feat.CircularFingerprint(size=16)
>>> loader = dc.data.SDFLoader(["LogP(RRCK)"], featurizer=featurizer, sanitize=True)
>>> dataset = loader.create_dataset(os.path.join(current_dir, "tests", "membrane_
↪permeability.sdf"))
>>> len(dataset)
2
```

__init__(*tasks: List[str]*, *featurizer: Featurizer*, *sanitize: bool = False*, *log_every_n: int = 1000*)

Initialize SDF Loader

Parameters

- **tasks** (*list[str]*) – List of tasknames. These will be loaded from the SDF file.
- **featurizer** (Featurizer) – Featurizer to use to process data
- **sanitize** (*bool, optional (default False)*) – Whether to sanitize molecules.
- **log_every_n** (*int, optional (default 1000)*) – Writes a logging statement this often.

create_dataset(*inputs: Any | Sequence[Any], data_dir: str | None = None, shard_size: int | None = 8192*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided sdf files.

Parameters

- **inputs** (*List*) – List of inputs to process. Entries can be filenames or arbitrary objects. Each file should be supported format (.sdf) or compressed folder of .sdf files
- **data_dir** (*str, optional (default None)*) – Directory to store featurized dataset.
- **shard_size** (*int, optional (default 8192)*) – Number of examples stored in each shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type

DiskDataset

FASTALoader

class FASTALoader(*featurizer: Featurizer | None = None, auto_add_annotations: bool = False, legacy: bool = True*)

Handles loading of FASTA files.

FASTA files are commonly used to hold sequence data. This class provides convenience files to lead FASTA data and one-hot encode the genomic sequences for use in downstream learning tasks.

__init__(*featurizer: Featurizer | None = None, auto_add_annotations: bool = False, legacy: bool = True*)

Initialize FASTALoader.

Parameters

- **featurizer** (*Featurizer (default: None)*) – The Featurizer to be used for the loaded FASTA data.

  If featurizer is None and legacy is True, the original featurization logic is used, creating a one hot encoding of all included FASTA strings of shape (number of FASTA sequences, number of channels + 1, sequence length, 1).

  If featurizer is None and legacy is False, the featurizer is initialized as a OneHotFeaturizer object with charset (“A”, “C”, “T”, “G”) and max_length = None.

- **auto_add_annotations** (*bool (default False)*) – Whether create_dataset will automatically add [CLS] and [SEP] annotations to the sequences it reads in order to assist tokenization. Keep False if your FASTA file already includes [CLS] and [SEP] annotations.
- **legacy** (*bool (default True)*) – Whether to use legacy logic for featurization. Legacy mode will create a one hot encoding of the FASTA content of shape (number of FASTA sequences, number of channels + 1, max length, 1).

  Legacy mode is only tested for ACTGN charsets, and will be deprecated.

create_dataset(*input_files: str | Sequence[str], data_dir: str | None = None, shard_size: int | None = None*) → *DiskDataset*

Creates a *Dataset* from input FASTA files.

At present, FASTA support is limited and doesn’t allow for sharding.

Parameters

- **input_files** (*List[str]*) – List of fasta files.
- **data_dir** (*str, optional (default None)*) – Name of directory where featurized data is stored.
- **shard_size** (*int, optional (default None)*) – For now, this argument is ignored and each FASTA file gets its own shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type

DiskDataset

FASTQLoader

class FASTQLoader(*featurizer: Featurizer | None = None*, *auto_add_annotations: bool = False*, *return_quality_scores: bool = False*)

Handles loading of FASTQ files.

FASTQ files are commonly used to hold very large sequence data. It is a variant of FASTA format. This class provides convenience files to load FASTQ data and one-hot encode the genomic sequences for use in downstream learning tasks.

Example

```
>>> import os
>>> from deepchem.feat.molecule_featurizers import OneHotFeaturizer
>>> from deepchem.data.data_loader import FASTQLoader
>>> current_dir = os.path.dirname(os.path.abspath(__file__))
>>> input_file = os.path.join(current_dir, "tests", "sample1.fastq")
>>> loader = FASTQLoader()
>>> sequences = loader.create_dataset(input_file)
```

See also:

Info on the structure of FASTQ files <https://support.illumina.com/bulletins/2016/04/fastq-files-explained.html>

__init__(*featurizer: Featurizer | None = None*, *auto_add_annotations: bool = False*, *return_quality_scores: bool = False*)

Initialize FASTQLoader.

Parameters

- **featurizer** (*Featurizer (default: None)*) – The Featurizer to be used for the loaded FASTQ data. The featurizer is initialized as a OneHotFeaturizer object with charset (“A”, “C”, “T”, “G”) and max_length = None.
- **auto_add_annotations** (*bool (default False)*) – Whether create_dataset will automatically add [CLS] and [SEP] annotations to the sequences it reads in order to assist tokenization. Keep False if your FASTQ file already includes [CLS] and [SEP] annotations.
- **return_quality_scores** (*bool (default True)*) – returns the quality (likelihood) score of the nucleotides in the sequence.

create_dataset(*input_files: str | Sequence[str]*, *data_dir: str | None = None*, *shard_size: int | None = 4096*) → *DiskDataset*

Creates a *Dataset* from input FASTQ files.

Parameters

- **input_files** (*List[str]*) – List of fastQ files.
- **data_dir** (*str, optional (default None)*) – Name of directory where featurized data is stored. shard_size: int, optional (default 4096)

Returns
A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type
DiskDataset

InMemoryLoader

The `dc.data.InMemoryLoader` is designed to facilitate the processing of large datasets where you already hold the raw data in-memory (say in a pandas dataframe).

class InMemoryLoader(*tasks: List[str]*, *featurizer: Featurizer*, *id_field: str | None = None*, *log_every_n: int = 1000*)

Facilitate Featurization of In-memory objects.

When featurizing a dataset, it's often the case that the initial set of data (pre-featurization) fits handily within memory. (For example, perhaps it fits within a column of a pandas DataFrame.) In this case, it would be convenient to directly be able to featurize this column of data. However, the process of featurization often generates large arrays which quickly eat up available memory. This class provides convenient capabilities to process such in-memory data by checkpointing generated features periodically to disk.

Example

Here's an example with only datapoints and no labels or weights.

```
>>> import deepchem as dc
>>> smiles = ["C", "CC", "CCC", "CCCC"]
>>> featurizer = dc.feat.CircularFingerprint()
>>> loader = dc.data.InMemoryLoader(tasks=["task1"], featurizer=featurizer)
>>> dataset = loader.create_dataset(smiles, shard_size=2)
>>> len(dataset)
4
```

Here's an example with both datapoints and labels

```
>>> import deepchem as dc
>>> smiles = ["C", "CC", "CCC", "CCCC"]
>>> labels = [1, 0, 1, 0]
>>> featurizer = dc.feat.CircularFingerprint()
>>> loader = dc.data.InMemoryLoader(tasks=["task1"], featurizer=featurizer)
>>> dataset = loader.create_dataset(zip(smiles, labels), shard_size=2)
>>> len(dataset)
4
```

Here's an example with datapoints, labels, weights and ids all provided.

```
>>> import deepchem as dc
>>> smiles = ["C", "CC", "CCC", "CCCC"]
>>> labels = [1, 0, 1, 0]
>>> weights = [1.5, 0, 1.5, 0]
>>> ids = ["C", "CC", "CCC", "CCCC"]
>>> featurizer = dc.feat.CircularFingerprint()
>>> loader = dc.data.InMemoryLoader(tasks=["task1"], featurizer=featurizer)
>>> dataset = loader.create_dataset(zip(smiles, labels, weights, ids), shard_size=2)
>>> len(dataset)
4
```

create_dataset(*inputs: Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *DiskDataset*

Creates and returns a *Dataset* object by featurizing provided files.

Reads in *inputs* and uses *self.featurizer* to featurize the data in these input files. For large files, automatically shards into smaller chunks of *shard_size* datapoints for convenience. Returns a *Dataset* object that contains the featurized dataset.

This implementation assumes that the helper methods *_get_shards* and *_featurize_shard* are implemented and that each shard returned by *_get_shards* is a pandas dataframe. You may choose to reuse or override this method in your subclass implementations.

Parameters

- **inputs** (*Sequence[Any]*) – List of inputs to process. Entries can be arbitrary objects so long as they are understood by *self.featurizer*
- **data_dir** (*str, optional (default None)*) – Directory to store featurized dataset.
- **shard_size** (*int, optional (default 8192)*) – Number of examples stored in each shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type

DiskDataset

__init__(*tasks: List[str]*, *featurizer:* Featurizer, *id_field: str | None = None*, *log_every_n: int = 1000*)

Construct a DataLoader object.

This constructor is provided as a template mainly. You shouldn't ever call this constructor directly as a user.

Parameters

- **tasks** (*List[str]*) – List of task names
- **featurizer** (Featurizer) – Featurizer to use to process data.
- **id_field** (*str, optional (default None)*) – Name of field that holds sample identifier. Note that the meaning of "field" depends on the input data type and can have a different meaning in different subclasses. For example, a CSV file could have a field as a column, and an SDF file could have a field as molecular property.
- **log_every_n** (*int, optional (default 1000)*) – Writes a logging statement this often.

Density Functional Theory YAML Loader

class DFTYamlLoader

Creates a *Dataset* object from YAML input files.

This class provides methods to load and featurize data from a YAML file. Although, in this class, we only focus on a specfic input format that can be used to perform Density Functional Theory calculations.

Examples

```
>>> from deepchem.data.data_loader import DFTYamlLoader
>>> import deepchem as dc
>>> import pytest
>>> inputs = 'deepchem/data/tests/dftdata.yaml'
>>> data = DFTYamlLoader()
>>> output = data.create_dataset(inputs)
```

Notes

Format (and example) for the YAML file:

- e_type : 'ae' true_val : '0.09194410469' systems : [{

    'moldesc': 'Li 1.5070 0 0; H -1.5070 0 0', 'basis': '6-311++G(3df,3pd)'

  }]

Each entry in the YAML file must contain the three parameters : e_type, true_val and systems in this particular order. One entry object may contain one or more systems. This data class does not support/ require an additional featurizer, since the datapoints are featurized within the methods. To read more about the parameters and their possible values please refer to deepchem.feat.dft_data.

__init__()

Initialize DFTYAML loader

create_dataset(*inputs: Any | Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 1*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided YAML files.

Parameters

- **input_files** (*OneOrMany[str]*) – List of YAML filenames.
- **data_dir** (*Optional[str], default None*) – Name of directory where featurized data is stored.
- **shard_size** (*int, optional (default 1)*) – Shard size when loading data.

Returns

A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type

DiskDataset

SAM Loader

class SAMLoader(*featurizer: Featurizer | None = None*)

Handles loading of SAM files. Sequence Alignment Map (SAM) is a text-based format used for storing biological sequences aligned to a reference sequence.It is generally used for storing nucleotide sequences, generated by next generation sequencing technologies, and unmapped sequences. SAM files have a header section and an alignment section.Alignment sections have 11 mandatory fields, as well as a variable number of optional fields. Here, we extract Query Name, Query Sequence, Query Length, Reference Name, Reference Start, CIGAR and Mapping Quality of each read in the SAM file. This class provides methods to load and featurize data from SAM files.

Examples

```
>>> from deepchem.data.data_loader import SAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.sam'
>>> data = SAMLoader()
>>> output = data.create_dataset(inputs)
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*featurizer: Featurizer | None = None*)

Initialize SAMLoader.

Parameters

featurizer (`Featurizer (default: None)`) – The Featurizer to be used for the loaded SAM data.

create_dataset(*input_files: str | Sequence[str], data_dir: str | None = None, shard_size: int | None = None*) → *DiskDataset*

Creates a *Dataset* from input SAM files.

Parameters

- **input_files** (`List[str]`) – List of SAM files.
- **data_dir** (`str, optional (default None)`) – Name of directory where featurized data is stored.
- **shard_size** (`int, optional (default None)`) – For now, this argument is ignored and each SAM file gets its own shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type

DiskDataset

BAM Loader

class BAMLoader(*featurizer: Featurizer | None = None*)

Handles loading of BAM files. Binary Alignment Map (BAM) is the comprehensive raw data of genome sequencing. It consists of the lossless, compressed binary representation of the Sequence Alignment Map files. BAM files are smaller and more efficient to work with than SAM files, saving time and reducing costs of computation and storage. BAM files store alignment data and often have corresponding BAM index files.The structure of BAM files include a header section and an alignment section. Here, we extract Query Name, Query Sequence, Query Length, Reference Name, Reference Start, CIGAR and Mapping Quality of each read in the BAM file. This class provides methods to load and featurize data from BAM files.

Examples

```
>>> from deepchem.data.data_loader import BAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.bam'
>>> data = BAMLoader()
>>> output = data.create_dataset(inputs)
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*featurizer: Featurizer | None = None*)

Initialize BAMLoader.

Parameters

featurizer (*Featurizer (default: None)*) – The Featurizer to be used for the loaded BAM data.

create_dataset(*input_files: str | Sequence[str], data_dir: str | None = None, shard_size: int | None = None*) → *DiskDataset*

Creates a *Dataset* from input BAM files.

Parameters

- **input_files** (*List[str]*) – List of BAM files, with their corresponding index files.
- **data_dir** (*str, optional (default None)*) – Name of directory where featurized data is stored.
- **shard_size** (*int, optional (default None)*) – For now, this argument is ignored and each BAM file gets its own shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type

DiskDataset

CRAM Loader

class CRAMLoader(*featurizer: Featurizer | None = None*)

Handles loading of CRAM files. Compressed Reference-oriented Alignment Map (CRAM) is a compressed columnar file format for storing biological sequences aligned to a reference sequence. CRAM is an efficient reference-based alternative to the Sequence Alignment Map (SAM) and Binary Alignment Map (BAM) file formats. The basic structure of a CRAM file has a series of containers, the first of which holds a compressed copy of the SAM header. Subsequent containers consist of a container Compression Header followed by a series of slices which hold the alignment records, formatted as a series of blocks. Here, we extract Query Name, Query Sequence, Query Length, Reference Name, Reference Start, CIGAR and Mapping Quality of each read in the CRAM file. This class provides methods to load and featurize data from CRAM files.

Examples

```
>>> from deepchem.data.data_loader import CRAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.cram'
>>> data = CRAMLoader()
>>> output = data.create_dataset(inputs)
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*featurizer: Featurizer | None = None*)

Initialize CRAMLoader.

Parameters

featurizer (`Featurizer (default: None)`) – The Featurizer to be used for the loaded CRAM data.

create_dataset(*input_files: str | Sequence[str], data_dir: str | None = None, shard_size: int | None = None*) → *DiskDataset*

Creates a *Dataset* from input CRAM files.

Parameters

- **input_files** (`List[str]`) – List of CRAM files.
- **data_dir** (`str, optional (default None)`) – Name of directory where featurized data is stored.
- **shard_size** (`int, optional (default None)`) – For now, this argument is ignored and each CRAM file gets its own shard.

Returns

A *DiskDataset* object containing a featurized representation of data from *input_files*.

Return type

DiskDataset

3.8.3 Data Classes

DeepChem featurizers often transform members into "data classes". These are classes that hold all the information needed to train a model on that data point. Models then transform these into the tensors for training in their `default_generator` methods.

Graph Data

These classes document the data classes for graph convolutions. We plan to simplify these classes (`ConvMol`, `MultiConvMol`, `WeaveMol`) into a joint data representation (`GraphData`) for all graph convolutions in a future version of DeepChem, so these APIs may not remain stable.

The graph convolution models which inherit `KerasModel` depend on `ConvMol`, `MultiConvMol`, or `WeaveMol`. On the other hand, the graph convolution models which inherit `TorchModel` depend on `GraphData`.

class ConvMol(*atom_features*, *adj_list*, *max_deg=10*, *min_deg=0*)

Holds information about a molecules.

Resorts order of atoms internally to be in order of increasing degree. Note that only heavy atoms (hydrogens excluded) are considered here.

__init__(*atom_features*, *adj_list*, *max_deg=10*, *min_deg=0*)

Parameters

- **atom_features** (*np.ndarray*) – Has shape (n_atoms, n_feat)
- **adj_list** (*list*) – List of length n_atoms, with neighor indices of each atom.
- **max_deg** (*int, optional*) – Maximum degree of any atom.
- **min_deg** (*int, optional*) – Minimum degree of any atom.

get_atoms_with_deg(*deg*)

Retrieves atom_features with the specific degree

get_num_atoms_with_deg(*deg*)

Returns the number of atoms with the given degree

get_atom_features()

Returns canonicalized version of atom features.

Features are sorted by atom degree, with original order maintained when degrees are same.

get_adjacency_list()

Returns a canonicalized adjacency list.

Canonicalized means that the atoms are re-ordered by degree.

Returns
Canonicalized form of adjacency list.

Return type
list

get_deg_adjacency_lists()

Returns adjacency lists grouped by atom degree.

Returns
Has length (max_deg+1-min_deg). The element at position deg is itself a list of the neighbor-lists for atoms with degree deg.

Return type
list

`get_deg_slice()`

Returns degree-slice tensor.

The deg_slice tensor allows indexing into a flattened version of the molecule's atoms. Assume atoms are sorted in order of degree. Then deg_slice[deg][0] is the starting position for atoms of degree deg in flattened list, and deg_slice[deg][1] is the number of atoms with degree deg.

Note deg_slice has shape (max_deg+1-min_deg, 2).

Returns
deg_slice – Shape (max_deg+1-min_deg, 2)

Return type
np.ndarray

`static get_null_mol(`*n_feat*, *max_deg=10*, *min_deg=0*`)`

Constructs a null molecules

Get one molecule with one atom of each degree, with all the atoms connected to themselves, and containing n_feat features.

Parameters
n_feat (*int*) – number of features for the nodes in the null molecule

`static agglomerate_mols(`*mols*, *max_deg=10*, *min_deg=0*`)`

Concatenates list of ConvMol's into one mol object that can be used to feed
into tensorflow placeholders. The indexing of the molecules are preseved during the combination, but the indexing of the atoms are greatly changed.

Parameters
mols (*list*) – ConvMol objects to be combined into one molecule.

`class MultiConvMol(`*nodes*, *deg_adj_lists*, *deg_slice*, *membership*, *num_mols*`)`

Holds information about multiple molecules, for use in feeding information into tensorflow. Generated using the agglomerate_mols function

`__init__(`*nodes*, *deg_adj_lists*, *deg_slice*, *membership*, *num_mols*`)`

`get_deg_adjacency_lists()`

`get_atom_features()`

`get_num_atoms()`

`get_num_molecules()`

`__module__ = 'deepchem.feat.mol_graphs'`

`class WeaveMol(`*nodes*, *pairs*, *pair_edges*`)`

Molecular featurization object for weave convolutions.

These objects are produced by WeaveFeaturizer, and feed into WeaveModel. The underlying implementation is inspired by[1].

[1] Kearnes, Steven, et al. "Molecular graph convolutions: moving beyond fingerprints." Journal of computer-aided molecular design 30.8 (2016): 595-608.

References

`__init__`(*nodes*, *pairs*, *pair_edges*)

`get_pair_edges`()

`get_pair_features`()

`get_atom_features`()

`get_num_atoms`()

`get_num_features`()

`__module__ = 'deepchem.feat.mol_graphs'`

class `GraphData`(*node_features: ndarray*, *edge_index: ndarray*, *edge_features: ndarray | None = None*, *node_pos_features: ndarray | None = None*, ***kwargs*)

GraphData class

This data class is almost same as torch_geometric.data.Data.

`node_features`

Node feature matrix with shape [num_nodes, num_node_features]

Type
np.ndarray

`edge_index`

Graph connectivity in COO format with shape [2, num_edges]

Type
np.ndarray, dtype int

`edge_features`

Edge feature matrix with shape [num_edges, num_edge_features]

Type
np.ndarray, optional (default None)

`node_pos_features`

Node position matrix with shape [num_nodes, num_dimensions].

Type
np.ndarray, optional (default None)

`num_nodes`

The number of nodes in the graph

Type
int

`num_node_features`

The number of features per node in the graph

Type
int

num_edges

The number of edges in the graph

Type

int

num_edges_features

The number of features per edge in the graph

Type

int, optional (default None)

Examples

```
>>> import numpy as np
>>> node_features = np.random.rand(5, 10)
>>> edge_index = np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]], dtype=np.int64)
>>> edge_features = np.random.rand(5, 5)
>>> global_features = np.random.random(5)
>>> graph = GraphData(node_features, edge_index, edge_features, z=global_features)
>>> graph
GraphData(node_features=[5, 10], edge_index=[2, 5], edge_features=[5, 5], z=[5])
```

__init__(*node_features: ndarray*, *edge_index: ndarray*, *edge_features: ndarray | None = None*, *node_pos_features: ndarray | None = None*, ***kwargs*)

Parameters

- **node_features** (*np.ndarray*) – Node feature matrix with shape [num_nodes, num_node_features]
- **edge_index** (*np.ndarray, dtype int*) – Graph connectivity in COO format with shape [2, num_edges]
- **edge_features** (*np.ndarray, optional (default None)*) – Edge feature matrix with shape [num_edges, num_edge_features]
- **node_pos_features** (*np.ndarray, optional (default None)*) – Node position matrix with shape [num_nodes, num_dimensions].
- **kwargs** (*optional*) – Additional attributes and their values

to_pyg_graph()

Convert to PyTorch Geometric graph data instance

Returns

Graph data for PyTorch Geometric

Return type

torch_geometric.data.Data

Note: This method requires PyTorch Geometric to be installed.

to_dgl_graph(*self_loop: bool = False*)

Convert to DGL graph data instance

Returns

- *dgl.DGLGraph* – Graph data for DGL
- **self_loop** (*bool*) – Whether to add self loops for the nodes, i.e. edges from nodes to themselves. Default to False.

Note: This method requires DGL to be installed.

numpy_to_torch(*device: str = 'cpu'*)

Convert numpy arrays to torch tensors. This may be useful when you are using PyTorch Geometric with GraphData objects.

Parameters
device (*str*) – Device to store the tensors. Default to 'cpu'.

Example

```
>>> num_nodes, num_node_features = 5, 32
>>> num_edges, num_edge_features = 6, 32
>>> node_features = np.random.random_sample((num_nodes, num_node_features))
>>> edge_features = np.random.random_sample((num_edges, num_edge_features))
>>> edge_index = np.random.randint(0, num_nodes, (2, num_edges))
>>> graph_data = GraphData(node_features, edge_index, edge_features)
>>> graph_data = graph_data.numpy_to_torch()
>>> print(type(graph_data.node_features))
<class 'torch.Tensor'>
```

subgraph(*nodes*)

Returns a subgraph of *nodes* indicies.

Parameters
nodes (*list, iterable*) – A list of node indices to be included in the subgraph.

Returns
subgraph_data – A new GraphData object containing the subgraph induced on *nodes*.

Return type
GraphData

Example

```
>>> import numpy as np
>>> from deepchem.feat.graph_data import GraphData
>>> node_features = np.random.rand(5, 10)
>>> edge_index = np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]], dtype=np.int64)
>>> edge_features = np.random.rand(5, 3)
>>> graph_data = GraphData(node_features, edge_index, edge_features)
>>> nodes = [0, 2, 4]
>>> subgraph_data, node_mapping = graph_data.subgraph(nodes)
```

Density Functional Theory Data

These Data classes are used to create entry objects for DFT calculations.

3.8.4 Base Classes (for develop)

Dataset

The `dc.data.Dataset` class is the abstract parent class for all datasets. This class should never be directly initialized, but contains a number of useful method implementations.

class Dataset

Abstract base class for datasets defined by X, y, w elements.

Dataset objects are used to store representations of a dataset as used in a machine learning task. Datasets contain features *X*, labels *y*, weights *w* and identifiers *ids*. Different subclasses of *Dataset* may choose to hold *X, y, w, ids* in memory or on disk.

The *Dataset* class attempts to provide for strong interoperability with other machine learning representations for datasets. Interconversion methods allow for *Dataset* objects to be converted to and from numpy arrays, pandas dataframes, tensorflow datasets, and pytorch datasets (only to and not from for pytorch at present).

Note that you can never instantiate a *Dataset* object directly. Instead you will need to instantiate one of the concrete subclasses.

__init__() → None

__len__() → int

Get the number of elements in the dataset.

Returns
The number of elements in the dataset.

Return type
int

get_shape() → Tuple[Tuple[int, ...], Tuple[int, ...], Tuple[int, ...], Tuple[int, ...]]

Get the shape of the dataset.

Returns four tuples, giving the shape of the X, y, w, and ids arrays.

Returns
The tuple contains four elements, which are the shapes of the X, y, w, and ids arrays.

Return type
Tuple

get_task_names() → ndarray

Get the names of the tasks associated with this dataset.

property X: ndarray

Get the X vector for this dataset as a single numpy array.

Returns
A numpy array of identifiers *X*.

Return type
np.ndarray

Note: If data is stored on disk, accessing this field may involve loading data from disk and could potentially be slow. Using *iterbatches()* or *itersamples()* may be more efficient for larger datasets.

`property y: ndarray`

Get the y vector for this dataset as a single numpy array.

Returns
A numpy array of identifiers *y*.

Return type
np.ndarray

Note: If data is stored on disk, accessing this field may involve loading data from disk and could potentially be slow. Using *iterbatches()* or *itersamples()* may be more efficient for larger datasets.

`property ids: ndarray`

Get the ids vector for this dataset as a single numpy array.

Returns
A numpy array of identifiers *ids*.

Return type
np.ndarray

Note: If data is stored on disk, accessing this field may involve loading data from disk and could potentially be slow. Using *iterbatches()* or *itersamples()* may be more efficient for larger datasets.

`property w: ndarray`

Get the weight vector for this dataset as a single numpy array.

Returns
A numpy array of weights *w*.

Return type
np.ndarray

Note: If data is stored on disk, accessing this field may involve loading data from disk and could potentially be slow. Using *iterbatches()* or *itersamples()* may be more efficient for larger datasets.

`iterbatches`(*batch_size: int | None = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over minibatches from the dataset.

Each minibatch is returned as a tuple of four numpy arrays: *(X, y, w, ids)*.

Parameters

- **batch_size** (*int, optional (default None)*) – Number of elements in each batch.
- **epochs** (*int, optional (default 1)*) – Number of epochs to walk over dataset.
- **deterministic** (*bool, optional (default False)*) – If True, follow deterministic order.

- **pad_batches** (*bool, optional (default False)*) – If True, pad each batch to *batch_size*.

Returns
Generator which yields tuples of four numpy arrays *(X, y, w, ids)*.

Return type
Iterator[Batch]

itersamples() → Iterator[Tuple[ndarray, ndarray, ndarray, ndarray]]

Get an object that iterates over the samples in the dataset.

Examples

```
>>> dataset = NumpyDataset(np.ones((2,2)))
>>> for x, y, w, id in dataset.itersamples():
...   print(x.tolist(), y.tolist(), w.tolist(), id)
[1.0, 1.0] [0.0] [0.0] 0
[1.0, 1.0] [0.0] [0.0] 1
```

transform(*transformer: Transformer*, ***args*) → *Dataset*

Construct a new dataset by applying a transformation to every sample in this dataset.

The argument is a function that can be called as follows: >> newx, newy, neww = fn(x, y, w)

It might be called only once with the whole dataset, or multiple times with different subsets of the data. Each time it is called, it should transform the samples and return the transformed data.

Parameters
transformer (*dc.trans.Transformer*) – The transformation to apply to each sample in the dataset.

Returns
A newly constructed Dataset object.

Return type
Dataset

select(*indices: Sequence[int] | ndarray*, *select_dir: str | None = None*) → *Dataset*

Creates a new dataset from a selection of indices from self.

Parameters

- **indices** (*Sequence*) – List of indices to select.
- **select_dir** (*str, optional (default None)*) – Path to new directory that the selected indices will be copied to.

get_statistics(*X_stats: bool = True*, *y_stats: bool = True*) → Tuple[ndarray, ...]

Compute and return statistics of this dataset.

Uses *self.itersamples()* to compute means and standard deviations of the dataset. Can compute on large datasets that don't fit in memory.

Parameters

- **X_stats** (*bool, optional (default True)*) – If True, compute feature-level mean and standard deviations.
- **y_stats** (*bool, optional (default True)*) – If True, compute label-level mean and standard deviations.

Returns

- If *X_stats == True*, returns *(X_means, X_stds)*.
- If *y_stats == True*, returns *(y_means, y_stds)*.
- If both are true, returns *(X_means, X_stds, y_means, y_stds)*.

Return type
Tuple

`make_tf_dataset`(*batch_size: int = 100*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*)

Create a tf.data.Dataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w) for one batch.

Parameters

- **batch_size** (*int, default 100*) – The number of samples to include in each batch.
- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset.
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **pad_batches** (*bool, default False*) – If True, batches are padded as necessary to make the size of each batch exactly equal batch_size.

Returns
TensorFlow Dataset that iterates over the same data.

Return type
tf.data.Dataset

Note: This class requires TensorFlow to be installed.

`make_pytorch_dataset`(*epochs: int = 1*, *deterministic: bool = False*, *batch_size: int | None = None*)

Create a torch.utils.data.IterableDataset that iterates over the data in this Dataset.

Each value returned by the Dataset's iterator is a tuple of (X, y, w, id) containing the data for one batch, or for a single sample if batch_size is None.

Parameters

- **epochs** (*int, default 1*) – The number of times to iterate over the Dataset.
- **deterministic** (*bool, default False*) – If True, the data is produced in order. If False, a different random permutation of the data is used for each epoch.
- **batch_size** (*int, optional (default None)*) – The number of samples to return in each batch. If None, each returned value is a single sample.

Returns
torch.utils.data.IterableDataset that iterates over the data in this dataset.

Return type
torch.utils.data.IterableDataset

Note: This class requires PyTorch to be installed.

`to_dataframe()` → DataFrame

Construct a pandas DataFrame containing the data from this Dataset.

Returns

Pandas dataframe. If there is only a single feature per datapoint, will have column "X" else will have columns "X1,X2,…" for features. If there is only a single label per datapoint, will have column "y" else will have columns "y1,y2,…" for labels. If there is only a single weight per datapoint will have column "w" else will have columns "w1,w2,…". Will have column "ids" for identifiers.

Return type

pd.DataFrame

`static from_dataframe`(*df: DataFrame*, *X: str | Sequence[str] | None = None*, *y: str | Sequence[str] | None = None*, *w: str | Sequence[str] | None = None*, *ids: str | None = None*)

Construct a Dataset from the contents of a pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – The pandas DataFrame
- **X** (*str or List[str], optional (default None)*) – The name of the column or columns containing the X array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **y** (*str or List[str], optional (default None)*) – The name of the column or columns containing the y array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **w** (*str or List[str], optional (default None)*) – The name of the column or columns containing the w array. If this is None, it will look for default column names that match those produced by to_dataframe().
- **ids** (*str, optional (default None)*) – The name of the column containing the ids. If this is None, it will look for default column names that match those produced by to_dataframe().

`to_csv`(*path: str*) → None

Write object to a comma-seperated values (CSV) file

Example

```
>>> import numpy as np
>>> X = np.random.rand(10, 10)
>>> dataset = dc.data.DiskDataset.from_numpy(X)
>>> dataset.to_csv('out.csv')
```

Parameters

path (*str*) – File path or object

Return type

None

DataLoader

The `dc.data.DataLoader` class is the abstract parent class for all dataloaders. This class should never be directly initialized, but contains a number of useful method implementations.

`class DataLoader`(*tasks: List[str]*, *featurizer:* Featurizer, *id_field: str | None = None*, *log_every_n: int = 1000*)

Handles loading/featurizing of data from disk.

The main use of *DataLoader* and its child classes is to make it easier to load large datasets into *Dataset* objects.`

DataLoader is an abstract superclass that provides a general framework for loading data into DeepChem. This class should never be instantiated directly. To load your own type of data, make a subclass of *DataLoader* and provide your own implementation for the *create_dataset()* method.

To construct a *Dataset* from input data, first instantiate a concrete data loader (that is, an object which is an instance of a subclass of *DataLoader*) with a given *Featurizer* object. Then call the data loader's *create_dataset()* method on a list of input files that hold the source data to process. Note that each subclass of *DataLoader* is specialized to handle one type of input data so you will have to pick the loader class suitable for your input data type.

Note that it isn't necessary to use a data loader to process input data. You can directly use *Featurizer* objects to featurize provided input into numpy arrays, but note that this calculation will be performed in memory, so you will have to write generators that walk the source files and write featurized data to disk yourself. *DataLoader* and its subclasses make this process easier for you by performing this work under the hood.

`__init__`(*tasks: List[str]*, *featurizer:* Featurizer, *id_field: str | None = None*, *log_every_n: int = 1000*)

Construct a DataLoader object.

This constructor is provided as a template mainly. You shouldn't ever call this constructor directly as a user.

Parameters

- **tasks** (*List[str]*) – List of task names
- **featurizer** (Featurizer) – Featurizer to use to process data.
- **id_field** (*str, optional (default None)*) – Name of field that holds sample identifier. Note that the meaning of "field" depends on the input data type and can have a different meaning in different subclasses. For example, a CSV file could have a field as a column, and an SDF file could have a field as molecular property.
- **log_every_n** (*int, optional (default 1000)*) – Writes a logging statement this often.

`featurize`(*inputs: Any | Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *Dataset*

Featurize provided files and write to specified location.

DEPRECATED: This method is now a wrapper for *create_dataset()* and calls that method under the hood.

For large datasets, automatically shards into smaller chunks for convenience. This implementation assumes that the helper methods *_get_shards* and *_featurize_shard* are implemented and that each shard returned by *_get_shards* is a pandas dataframe. You may choose to reuse or override this method in your subclass implementations.

Parameters

- **inputs** (*List*) – List of inputs to process. Entries can be filenames or arbitrary objects.
- **data_dir** (*str, default None*) – Directory to store featurized dataset.
- **shard_size** (*int, optional (default 8192)*) – Number of examples stored in each shard.

Returns
A *Dataset* object containing a featurized representation of data from *inputs*.

Return type
Dataset

create_dataset(*inputs: Any | Sequence[Any]*, *data_dir: str | None = None*, *shard_size: int | None = 8192*) → *Dataset*

Creates and returns a *Dataset* object by featurizing provided files.

Reads in *inputs* and uses *self.featurizer* to featurize the data in these inputs. For large files, automatically shards into smaller chunks of *shard_size* datapoints for convenience. Returns a *Dataset* object that contains the featurized dataset.

This implementation assumes that the helper methods *_get_shards* and *_featurize_shard* are implemented and that each shard returned by *_get_shards* is a pandas dataframe. You may choose to reuse or override this method in your subclass implementations.

Parameters

- **inputs** (*List*) – List of inputs to process. Entries can be filenames or arbitrary objects.
- **data_dir** (*str, optional (default None)*) – Directory to store featurized dataset.
- **shard_size** (*int, optional (default 8192)*) – Number of examples stored in each shard.

Returns
A *DiskDataset* object containing a featurized representation of data from *inputs*.

Return type
DiskDataset

3.9 MoleculeNet

The DeepChem library is packaged alongside the MoleculeNet suite of datasets. One of the most important parts of machine learning applications is finding a suitable dataset. The MoleculeNet suite has curated a whole range of datasets and loaded them into DeepChem `dc.data.Dataset` objects for convenience.

3.9.1 MoleculeNet Cheatsheet

When training a model or performing a benchmark, the user needs specific datasets. However, at the beginning, this search can be exhaustive and confusing. The following cheatsheet is aimed at helping DeepChem users identify more easily which dataset to use depending on their purposes.

Each row reprents a dataset where a brief description is given. Also, the columns represents the type of the data; depending on molecule properties, images or materials and how many data points they have. Each dataset is referenced with a link of the paper. Finally, there are some entries that need further information.

Cheatsheet

Table 1:

Name	Description
BACE (Regression)	Provides bindings results for a set of inhibitors of human beta-secretase (BACE-1)
BACE (Classification)	Provides bindings results for a set of inhibitors of human beta-secretase (BACE-1)

Table 1 – co

Name	Description
BBBC (BBBC001)	Images of HT29 colon cancer cells
BBBC (BBBC002)	Images of Drosophilia Kc167 cells
BBBC (BBBC003)	DIC Images of Mouse Embryos
BBBC (BBBC004)	Synthetic Images of clustered nuclei
BBBC (BBBC004)	Synthetic Images of clustered nuclei
BBBP	Blood-Brain Barrier Penetration designed for the modeling and prediction of barrier permeability
Cell Counting	Synthetic emulations of fluorescence microscopic images of bacterial cells
ChEMBL (set = ‘sparse’)	A sparse subset of ChEMBL with activity data for one target
ChEMBL (set = ‘5thresh’)	A subset of ChEMBL with activity data for at least five targets
ChEMBL25	
Clearance	
Clintox	Compares drugs approved by the FDA and drugs that have failed clinical trials for toxicity reasons.
Delaney	A regression dataset containing structures and water solubility data
Factors	Merck in-house compounds that were measured for IC50 of inhibition on 12 serine proteases
Freesolv	A collection of experimental and calculated hydration free energies for small molecules in water
HIV	A dataset wich tested the ability to inhibit HIV replication
HOPV	Harvard Organic Photovoltaic dataset utilized as p-type materials
HPPB	Thermosynamic solubility datasets
KAGGLE	in-house compounds that were measured on 15 enzyme inhibition and ADME/TOX datasets.
KINASE	In-house compounds that were measured for IC50 of inhibition on 99 protein kinases
LIPO	Experimental results of octanol/water distribution coefficient (logD at pH 7.4)
Band Gap	Experimentally measured band gaps for inorganic crystal structure
Perovskite	Contains Perovskite structures and their formation energies
MP Formation Energy	Contains calculated formation energies and inorganic crystal structures from the Materials Project databa
MP Metallicity	Contains inorganic crystal structures from the Materials Project database labeled as metals or nonmetals
MUV	Benchmark dataset selected from PubChem BioAssay by applying a refined nearest neighbor analysis
NCI	
PCBA	Database consisting of biological activities of small molecules generated by high-throughput screening
PDBBIND	Experimental binding affinity data and structures of protein-ligand complexes
PPB	
QM7	Subset of GDB-13 containing up to 7 heavy atoms CNOS
QM8	Dataset used in a study on modeling quantum mechanical calculations of electronic spectra and excited s
QM9	Dataset that provides geometric/energetic/electronic and thermodynamic properties for a subset of GDB-
SAMPL	Similat to FreeSolv dataset which provides experimental and calculated hydration free energy of small m
SIDER	The Side Effect Resource (SIDER) is a database of marketed drugs and adverse drug reactions (ADR)
Thermosol	Thermodynamic solubility datasets
Tox21	The “Toxicology in the 21st Century” (Tox21) initiative created a public database measuring the toxicity
Toxcast	Toxicology data for an extensive library of compounds based on in vitro high-throughput screening
USPTO	Subsets of USPTO dataset of organic chemical reactions extracted from US patents and patent applicatio
UV	The UV dataset tests Merck’s internal compounds on 190 absorption wavelengths between 210 and 400 r
ZINC15	Purchasable compounds for virtual screening of small molecules to identify structures that are likely to b
Platinum Adsorption	Different configurations of Adsorbates (i.e N and NO) on Platinum surface represented as Lattice and the

3.9.2 Contributing a new dataset to MoleculeNet

If you are proposing a new dataset to be included in the MoleculeNet benchmarking suite, please follow the instructions below. Please review the datasets already available in MolNet before contributing.

0. Read the Contribution guidelines.
1. Open an issue to discuss the dataset you want to add to MolNet.
2. Write a *DatasetLoader* class that inherits from deepchem.molnet.load_function.molnet_loader._MolnetLoader and implements a *create_dataset* method. See the _QM9Loader for a simple example.
3. Write a *load_dataset* function that documents the dataset and add your load function to deepchem.molnet.__init__.py for easy importing.
4. Prepare your dataset as a .tar.gz or .zip file. Accepted filetypes include CSV, JSON, and SDF.
5. Ask a member of the technical steering committee to add your .tar.gz or .zip file to the DeepChem AWS bucket. Modify your load function to pull down the dataset from AWS.
6. Add documentation for your loader to the MoleculeNet docs.
7. Submit a [WIP] PR (Work in progress pull request) following the PR template.

3.9.3 Example Usage

Below is an example of how to load a MoleculeNet dataset and featurizer. This approach will work for any dataset in MoleculeNet by changing the load function and featurizer. For more details on the featurizers, see the *Featurizers* section.

```
import deepchem as dc
from deepchem.feat.molecule_featurizers import MolGraphConvFeaturizer

featurizer = MolGraphConvFeaturizer(use_edges=True)
dataset_dc = dc.molnet.load_qm9(featurizer=featurizer)
tasks, dataset, transformers = dataset_dc
train, valid, test = dataset

x,y,w,ids = train.X, train.y, train.w, train.ids
```

Note that the "w" matrix represents the weight of each sample. Some assays may have missing values, in which case the weight is 0. Otherwise, the weight is 1.

Additionally, the environment variable `DEEPCHEM_DATA_DIR` can be set like `os.environ['DEEPCHEM_DATA_DIR'] = path/to/store/featurized/dataset`. When the `DEEPCHEM_DATA_DIR` environment variable is set, molnet loader stores the featurized dataset in the specified directory and when the dataset has to be reloaded the next time, it will be fetched from the data directory directly rather than featurizing the raw dataset from scratch.

3.9.4 BACE Dataset

`load_bace_classification`(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BACE dataset with classification labels.

BACE dataset with classification labels ("class"). The BACE dataset contains 1513 compounds and the dataset is a binary classification dataset with labels 0 or 1.

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (`list of TransformerGenerators or strings`) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (`bool`) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (`str`) – a directory to save the raw data in
- **save_dir** (`str`) – a directory to save the dataset in

`load_bace_regression`(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BACE dataset, regression labels

The BACE dataset provides quantitative IC50 and qualitative (binary label) binding results for a set of inhibitors of human beta-secretase 1 (BACE-1).

All data are experimental values reported in scientific literature over the past decade, some with detailed crystal structures available. A collection of 1522 compounds is provided, along with the regression labels of IC50. The number of tasks in the dataset is one.

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "mol" - SMILES representation of the molecular structure
- "pIC50" - Negative log of the IC50 binding affinity
- "class" - Binary labels for inhibitor

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.

- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.5 BBBC Datasets

load_bbbc001(*splitter: Splitter | str | None = 'index', transformers: List[TransformerGenerator | str] = [], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBC001 dataset

This dataset contains 6 images of human HT29 colon cancer cells. The task is to learn to predict the cell counts in these images. This dataset is too small to serve to train algorithms, but might serve as a good test dataset. https://data.broadinstitute.org/bbbc/BBBC001/

Parameters

- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

load_bbbc002(*splitter: Splitter | str | None = 'index', transformers: List[TransformerGenerator | str] = [], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBC002 dataset

This dataset contains data corresponding to 5 samples of Drosophilia Kc167 cells. There are 10 fields of view for each sample, each an image of size 512x512. Ground truth labels contain cell counts for this dataset. Full details about this dataset are present at https://data.broadinstitute.org/bbbc/BBBC002/.

Parameters

- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.

- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

load_bbbc003(*load_segmentation_mask: bool = False*, *splitter: Splitter | str | None = 'index'*, *transformers: List[TransformerGenerator | str] = []*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBC003 dataset

This dataset contains data corresponding to 15 samples of Mouse embryos with DIC. Each image is of size 640x480. Ground truth labels contain cell counts and segmentation masks for this dataset. Full details about this dataset are present at https://data.broadinstitute.org/bbbc/BBBC003/.

Parameters

- **load_segmentation_mask** (*bool*) – if True, the dataset will contain segmentation masks as labels. Otherwise, the dataset will contain cell counts as labels.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Examples

Importing necessary modules

```
>>> import deepchem as dc
>>> import numpy as np
```

We can load the BBBC003 dataset with 2 types of labels: segmentation masks and cell counts. We will first load the dataset with cell counts as labels.

```
>>> loader = dc.molnet.load_bbbc003(load_segmentation_mask=False)
>>> tasks, dataset, transformers = loader
>>> train, val, test = dataset
```

We now have a dataset with 15 samples, each with 300 cells. The images are of size 640x480. The labels are cell counts. We can verify this as follows:

```
>>> train.X.shape
(12,)
>>> train.y.shape
(12,)
```

We will now load the dataset with segmentation masks as labels.

```
>>> loader = dc.molnet.load_bbbc003(load_segmentation_mask=True)
>>> tasks, dataset, transformers = loader
>>> train, val, test = dataset
```

We now have a dataset with 15 samples, each with 300 cells. The images are of size 640x480. The labels are segmentation masks. We can verify this as follows:

```
>>> print(train.X.shape)
(12,)
>>> print(train.y.shape)
(12,)
```

Note: The image labelled '7_19_M2E15.tif' is transposed to 480x640 in the source file along with it's segementation mask. To match it with the other images, we need to transpose it back to 640x480.

This image is found at index 6 in the train dataset (Assuming no shuffling has taken place).

First, we load the dataset as usual and split it into X, y, w and ids. Here, X is the list of input images, y is the list of labels, w is the list of weights and ids is the list of IDs for each sample.

```
>>> train_x, train_y, train_w, train_ids = train.X, train.y, train.w, train.ids
```

We can now transpose the image at index 6 in the input data (train_x): >>> train_x[6] = train_x[6].T

We can now verify that the image is of size 640x480: >>> print(train_x[6].shape) (640, 480)

This is also seen in the segmentation mask with the same filename and index, in which case, we transpose the label (train_y) instead of the input data:

```
>>> train_y[6] = train_y[6].T
```

We can now verify that the image is of size 640x480: >>> train_y[6].shape (640, 480)

load_bbbc004(*overlap_probability: float = 0.0*, *load_segmentation_mask: bool = False*, *splitter:* Splitter | *str* | *None = 'index'*, *transformers: List[TransformerGenerator | str] = []*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBC004 dataset

This dataset contains data corresponding to 20 samples of synthetically generated fluorescent cell population images. There are 300 cells in each sample, each an image of size 950x950. Ground truth labels contain cell counts and segmentation masks for this dataset. Full details about this dataset are present at https://data.broadinstitute.org/bbbc/BBBC004/.

Parameters

- **overlap_probability** (*float from list {0.0, 0.15, 0.3, 0.45, 0.6}*) – the overlap probability of the synthetic cells in the images
- **load_segmentation_mask** (*bool*) – if True, the dataset will contain segmentation masks as labels. Otherwise, the dataset will contain cell counts as labels.
- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.

- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Examples

Importing necessary modules

```
>>> import deepchem as dc
>>> import numpy as np
```

We can load the BBBC004 dataset with 2 types of labels: segmentation masks and cell counts. We will first load the dataset with cell counts as labels.

```
>>> loader = dc.molnet.load_bbbc004(overlap_probability=0.0, load_segmentation_
↪mask=False)
>>> tasks, dataset, transformers = loader
>>> train, val, test = dataset
```

We now have a dataset with 20 samples, each with 300 cells. The images are of size 950x950. The labels are cell counts. We can verify this as follows:

```
>>> train.X.shape
(16, 950, 950)
>>> train.y.shape
(16,)
```

We will now load the dataset with segmentation masks as labels.

```
>>> loader = dc.molnet.load_bbbc004(overlap_probability=0.0, load_segmentation_
↪mask=True)
>>> tasks, dataset, transformers = loader
>>> train, val, test = dataset
```

We now have a dataset with 20 samples, each with 300 cells. The images are of size 950x950. The labels are segmentation masks. We can verify this as follows:

```
>>> train.X.shape
(16, 950, 950)
>>> train.y.shape
(16, 950, 950, 3)
```

load_bbbc005(*splitter:* Splitter | *str* | *None* = *'index'*, *transformers: List[TransformerGenerator* | *str] = []*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBC005 dataset

This dataset contains data corresponding to 19,200 samples of synthetically generated fluorescent cell population images. These images were simulated for a given cell count with a clustering probablity of 25% and a CCD noise variance of 0.0001. Focus blur was simulated by applying varying Guassian filters to the images. Each image is of size 520x696. Ground truth labels contain cell counts for this dataset. Full details about this dataset are present at https://data.broadinstitute.org/bbbc/BBBC005/.

Parameters

- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Examples

Importing necessary modules

>> import deepchem as dc >> import numpy as np

We will now load the BBBC005 dataset with cell counts as labels.

>> loader = dc.molnet.load_bbbc005() >> tasks, dataset, transformers = loader >> train, val, test = dataset

We now have a dataset with a total of 19,200 samples with cell counts in the range of 1-100. The images are of size 520x696. The labels are cell counts. We have a train-val-test split of 80:10:10. We can verify this as follows:

>> train.X.shape (15360, 520, 696) >> train.y.shape (15360,)

3.9.6 BBBP Datasets

BBBP stands for Blood-Brain-Barrier Penetration

load_bbbp(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load BBBP dataset

The blood-brain barrier penetration (BBBP) dataset is designed for the modeling and prediction of barrier permeability. As a membrane separating circulating blood and brain extracellular fluid, the blood-brain barrier blocks most drugs, hormones and neurotransmitters. Thus penetration of the barrier forms a long-standing issue in development of drugs targeting central nervous system.

This dataset includes binary labels for over 2000 compounds on their permeability properties.

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "name" - Name of the compound
- "smiles" - SMILES representation of the molecular structure
- "p_np" - Binary labels for penetration/non-penetration

Parameters

- **featurizer** (Featurizer *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.

- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.7 Cell Counting Datasets

load_cell_counting(*splitter:* Splitter | *str* | *None = None*, *transformers: List[TransformerGenerator* | *str] = []*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Cell Counting dataset.

Loads the cell counting dataset from http://www.robots.ox.ac.uk/~vgg/research/counting/index_org.html.

Parameters

- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.8 Chembl Datasets

load_chembl(*featurizer:* Featurizer | *str = 'ECFP'*, *splitter:* Splitter | *str* | *None = 'scaffold'*, *transformers: List[TransformerGenerator* | *str] = ['normalization']*, *set: str = '5thresh'*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load the ChEMBL dataset.

This dataset is based on release 22.1 of the data from https://www.ebi.ac.uk/chembl/. Two subsets of the data are available, depending on the "set" argument. "sparse" is a large dataset with 244,245 compounds. As the name suggests, the data is extremely sparse, with most compounds having activity data for only one target. "5thresh" is a much smaller set (23,871 compounds) that includes only compounds with activity data for at least five targets.

Parameters

- **featurizer** (Featurizer *or* str) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or* str) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **set** (*str*) – the subset to load, either "sparse" or "5thresh"
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.9 Chembl25 Datasets

load_chembl25(*featurizer: Featurizer | str = 'ECFP'*, *splitter: Splitter | str | None = 'scaffold'*, *transformers: List[TransformerGenerator | str] = ['normalization']*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Loads the ChEMBL25 dataset, featurizes it, and does a split.

Parameters

- **featurizer** (Featurizer *or* str) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or* str) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.10 Clearance Datasets

load_clearance(*featurizer: Featurizer | str = 'ECFP'*, *splitter: Splitter | str | None = 'scaffold'*, *transformers: List[TransformerGenerator | str] = ['log']*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load clearance datasets.

Parameters

- **featurizer** (Featurizer *or* str) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.

- **splitter** (Splitter *or* str) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.11 Clintox Datasets

load_clintox(*featurizer:* Featurizer | *str = 'ECFP'*, *splitter:* Splitter | *str* | *None = 'scaffold'*, *transformers: List[TransformerGenerator* | *str] = ['balancing']*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load ClinTox dataset

The ClinTox dataset compares drugs approved by the FDA and drugs that have failed clinical trials for toxicity reasons. The dataset includes two classification tasks for 1491 drug compounds with known chemical structures:

1. clinical trial toxicity (or absence of toxicity)
2. FDA approval status.

List of FDA-approved drugs are compiled from the SWEETLEAD database, and list of drugs that failed clinical trials for toxicity reasons are compiled from the Aggregate Analysis of ClinicalTrials.gov(AACT) database.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- “smiles” - SMILES representation of the molecular structure
- “FDA_APPROVED” - FDA approval status
- “CT_TOX” - Clinical trial results

Parameters

- **featurizer** (Featurizer *or* str) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or* str) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.12 Delaney Datasets

load_delaney(*featurizer: Featurizer | str = 'ECFP'*, *splitter: Splitter | str | None = 'scaffold'*, *transformers: List[TransformerGenerator | str] = ['normalization']*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Delaney dataset

The Delaney (ESOL) dataset a regression dataset containing structures and water solubility data for 1128 compounds. The dataset is widely used to validate machine learning models on estimating solubility directly from molecular structures (as encoded in SMILES strings).

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "Compound ID" - Name of the compound
- "smiles" - SMILES representation of the molecular structure
- **"measured log solubility in mols per litre" - Log-scale water solubility**
  of the compound, used as label

Parameters

- **featurizer** (`Featurizer` *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter` *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.13 Factors Datasets

load_factors(*shard_size=2000*, *featurizer=None*, *split=None*, *reload=True*)

Loads FACTOR dataset; does not do train/test split

The Factors dataset is an in-house dataset from Merck that was first introduced in the following paper: Ramsundar, Bharath, et al. "Is multitask deep learning practical for pharma?." Journal of chemical information and modeling 57.8 (2017): 2068-2076.

It contains 1500 Merck in-house compounds that were measured for IC50 of inhibition on 12 serine proteases. Unlike most of the other datasets featured in MoleculeNet, the Factors collection does not have structures for the compounds tested since they were proprietary Merck compounds. However, the collection does feature pre-computed descriptors for these compounds.

Note that the original train/valid/test split from the source data was preserved here, so this function doesn't allow for alternate modes of splitting. Similarly, since the source data came pre-featurized, it is not possible to apply alternative featurizations.

Parameters

- **`shard_size`** (*`int, optional`*) – Size of the DiskDataset shards to write on disk
- **`featurizer`** (*`optional`*) – Ignored since featurization pre-computed
- **`split`** (*`optional`*) – Ignored since split pre-computed
- **`reload`** (*`bool, optional`*) – Whether to automatically re-load from disk

3.9.14 Freesolv Dataset

`load_freesolv`(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = MATFeaturizer[], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Freesolv dataset

The FreeSolv dataset is a collection of experimental and calculated hydration free energies for small molecules in water, along with their experiemental values. Here, we are using a modified version of the dataset with the molecule smile string and the corresponding experimental hydration free energies.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "mol" - SMILES representation of the molecular structure
- "y" - Experimental hydration free energy

Parameters

- **`featurizer`** (`Featurizer` *`or str`*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **`splitter`** (`Splitter` *`or str`*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **`transformers`** (*`list of TransformerGenerators or strings`*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **`reload`** (*`bool`*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **`data_dir`** (*`str`*) – a directory to save the raw data in
- **`save_dir`** (*`str`*) – a directory to save the dataset in

References

3.9.15 HIV Datasets

load_hiv(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load HIV dataset

The HIV dataset was introduced by the Drug Therapeutics Program (DTP) AIDS Antiviral Screen, which tested the ability to inhibit HIV replication for over 40,000 compounds. Screening results were evaluated and placed into three categories: confirmed inactive (CI),confirmed active (CA) and confirmed moderately active (CM). We further combine the latter two labels, making it a classification task between inactive (CI) and active (CA and CM).

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "smiles": SMILES representation of the molecular structure
- "activity": Three-class labels for screening results: CI/CM/CA
- "HIV_active": Binary labels for screening results: 1 (CA/CM) and 0 (CI)

Parameters

- **featurizer** (`Featurizer` *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter` *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.16 HOPV Datasets

HOPV stands for the Harvard Organic Photovoltaic Dataset.

load_hopv(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load HOPV datasets. Does not do train/test split

The HOPV datasets consist of the "Harvard Organic Photovoltaic Dataset. This dataset includes 350 small molecules and polymers that were utilized as p-type materials in OPVs. Experimental properties include: HOMO [a.u.], LUMO [a.u.], Electrochemical gap [a.u.], Optical gap [a.u.], Power conversion efficiency [%], Open circuit potential [V], Short circuit current density [mA/cm^2], and fill factor [%]. Theoretical calculations in the original dataset have been removed (for now).

Lopez, Steven A., et al. "The Harvard organic photovoltaic dataset." Scientific data 3.1 (2016): 1-7.

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.17 HPPB Datasets

load_hppb(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['log'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Loads the thermodynamic solubility datasets.

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.18 KAGGLE Datasets

`load_kaggle`(*shard_size=2000*, *featurizer=None*, *split=None*, *reload=True*)

Loads kaggle datasets. Generates if not stored already.

The Kaggle dataset is an in-house dataset from Merck that was first introduced in the following paper:

Ma, Junshui, et al. "Deep neural nets as a method for quantitative structure–activity relationships." Journal of chemical information and modeling 55.2 (2015): 263-274.

It contains 100,000 unique Merck in-house compounds that were measured on 15 enzyme inhibition and ADME/TOX datasets. Unlike most of the other datasets featured in MoleculeNet, the Kaggle collection does not have structures for the compounds tested since they were proprietary Merck compounds. However, the collection does feature pre-computed descriptors for these compounds.

Note that the original train/valid/test split from the source data was preserved here, so this function doesn't allow for alternate modes of splitting. Similarly, since the source data came pre-featurized, it is not possible to apply alternative featurizations.

Parameters

- **shard_size** (*int, optional*) – Size of the DiskDataset shards to write on disk
- **featurizer** (*optional*) – Ignored since featurization pre-computed
- **split** (*optional*) – Ignored since split pre-computed
- **reload** (*bool, optional*) – Whether to automatically re-load from disk

3.9.19 Kinase Datasets

`load_kinase`(*shard_size=2000*, *featurizer=None*, *split=None*, *reload=True*)

Loads Kinase datasets, does not do train/test split

The Kinase dataset is an in-house dataset from Merck that was first introduced in the following paper: Ramsundar, Bharath, et al. "Is multitask deep learning practical for pharma?." Journal of chemical information and modeling 57.8 (2017): 2068-2076.

It contains 2500 Merck in-house compounds that were measured for IC50 of inhibition on 99 protein kinases. Unlike most of the other datasets featured in MoleculeNet, the Kinase collection does not have structures for the compounds tested since they were proprietary Merck compounds. However, the collection does feature pre-computed descriptors for these compounds.

Note that the original train/valid/test split from the source data was preserved here, so this function doesn't allow for alternate modes of splitting. Similarly, since the source data came pre-featurized, it is not possible to apply alternative featurizations.

Parameters

- **shard_size** (*int, optional*) – Size of the DiskDataset shards to write on disk
- **featurizer** (*optional*) – Ignored since featurization pre-computed
- **split** (*optional*) – Ignored since split pre-computed
- **reload** (*bool, optional*) – Whether to automatically re-load from disk

3.9.20 Lipo Datasets

load_lipo(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Lipophilicity dataset

Lipophilicity is an important feature of drug molecules that affects both membrane permeability and solubility. The lipophilicity dataset, curated from ChEMBL database, provides experimental results of octanol/water distribution coefficient (logD at pH 7.4) of 4200 compounds.

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "smiles" - SMILES representation of the molecular structure
- **"exp" - Measured octanol/water distribution coefficient (logD) of the**
  compound, used as label

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.21 Materials Datasets

Materials datasets include inorganic crystal structures, chemical compositions, and target properties like formation energies and band gaps. Machine learning problems in materials science commonly include predicting the value of a continuous (regression) or categorical (classification) property of a material based on its chemical composition or crystal structure. "Inverse design" is also of great interest, in which ML methods generate crystal structures that have a desired property. Other areas where ML is applicable in materials include: discovering new or modified phenomenological models that describe material behavior

load_bandgap(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = ElementPropertyFingerprint[data_source='matminer'], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load band gap dataset.

Contains 4604 experimentally measured band gaps for inorganic crystal structure compositions. In benchmark studies, random forest models achieved a mean average error of 0.45 eV during five-fold nested cross validation on this dataset.

For more details on the dataset see [1]_. For more details on previous benchmarks for this dataset, see [2]_.

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

Examples

```
>>>
>> import deepchem as dc
>> tasks, datasets, transformers = dc.molnet.load_bandgap()
>> train_dataset, val_dataset, test_dataset = datasets
>> n_tasks = len(tasks)
>> n_features = train_dataset.get_data_shape()[0]
>> model = dc.models.MultitaskRegressor(n_tasks, n_features)
```

load_perovskite(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = DummyFeaturizer[], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load perovskite dataset.

Contains 18928 perovskite structures and their formation energies. In benchmark studies, random forest models and crystal graph neural networks achieved mean average error of 0.23 and 0.05 eV/atom, respectively, during five-fold nested cross validation on this dataset.

For more details on the dataset see [1]_. For more details on previous benchmarks for this dataset, see [2]_.

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

Examples

```
>>> import deepchem as dc
>>> tasks, datasets, transformers = dc.molnet.load_perovskite()
>>> train_dataset, val_dataset, test_dataset = datasets
>>> model = dc.models.CGCNNModel(mode='regression', batch_size=32, learning_rate=0.
→001)
```

`load_mp_formation_energy`(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load mp formation energy dataset.

Contains 132752 calculated formation energies and inorganic crystal structures from the Materials Project database. In benchmark studies, random forest models achieved a mean average error of 0.116 eV/atom during five-folded nested cross validation on this dataset.

For more details on the dataset see **[1]**_. For more details on previous benchmarks for this dataset, see **[2]**_.

Parameters

- **featurizer** (`Featurizer` *or* `str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter` *or* `str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

Examples

```
>>>
>> import deepchem as dc
>> tasks, datasets, transformers = dc.molnet.load_mp_formation_energy()
>> train_dataset, val_dataset, test_dataset = datasets
>> n_tasks = len(tasks)
>> n_features = train_dataset.get_data_shape()[0]
>> model = dc.models.MultitaskRegressor(n_tasks, n_features)
```

load_mp_metallicity(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load mp formation energy dataset.

Contains 106113 inorganic crystal structures from the Materials Project database labeled as metals or nonmetals. In benchmark studies, random forest models achieved a mean ROC-AUC of 0.9 during five-folded nested cross validation on this dataset.

For more details on the dataset see **[1]_**. For more details on previous benchmarks for this dataset, see **[2]_**.

Parameters

- **featurizer** (`Featurizer` *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter` *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

Examples

```
>>>
>> import deepchem as dc
>> tasks, datasets, transformers = dc.molnet.load_mp_metallicity()
>> train_dataset, val_dataset, test_dataset = datasets
>> n_tasks = len(tasks)
>> n_features = train_dataset.get_data_shape()[0]
>> model = dc.models.MultitaskRegressor(n_tasks, n_features)
```

3.9.22 MUV Datasets

load_muv(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load MUV dataset

The Maximum Unbiased Validation (MUV) group is a benchmark dataset selected from PubChem BioAssay by applying a refined nearest neighbor analysis.

The MUV dataset contains 17 challenging tasks for around 90 thousand compounds and is specifically designed for validation of virtual screening techniques.

Scaffold splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "mol_id" - PubChem CID of the compound
- "smiles" - SMILES representation of the molecular structure
- "MUV-XXX" - Measured results (Active/Inactive) for bioassays

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.23 NCI Datasets

load_nci(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'random', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load NCI dataset.

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (`list of TransformerGenerators or strings`) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (`bool`) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (`str`) – a directory to save the raw data in
- **save_dir** (`str`) – a directory to save the dataset in

3.9.24 PCBA Datasets

load_pcba(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load PCBA dataset

PubChem BioAssay (PCBA) is a database consisting of biological activities of small molecules generated by high-throughput screening. We use a subset of PCBA, containing 128 bioassays measured over 400 thousand compounds, used by previous work to benchmark machine learning methods.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "mol_id" - PubChem CID of the compound
- "smiles" - SMILES representation of the molecular structure
- **"PCBA-XXX" - Measured results (Active/Inactive) for bioassays:**
  search for the assay ID at https://pubchem.ncbi.nlm.nih.gov/search/#collection=bioassays for details

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.

- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.25 PDBBIND Datasets

load_pdbbind(*featurizer: ComplexFeaturizer, splitter: Splitter | str | None = 'random', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, pocket: bool = True, set_name: str = 'core', **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load PDBBind dataset.

The PDBBind dataset includes experimental binding affinity data and structures for 4852 protein-ligand complexes from the "refined set" and 12800 complexes from the "general set" in PDBBind v2019 and 193 complexes from the "core set" in PDBBind v2013. The refined set removes data with obvious problems in 3D structure, binding data, or other aspects and should therefore be a better starting point for docking/scoring studies. Details on the criteria used to construct the refined set can be found in [4]_. The general set does not include the refined set. The core set is a subset of the refined set that is not updated annually.

Random splitting is recommended for this dataset.

The raw dataset contains the columns below:

- "ligand" - SDF of the molecular structure
- "protein" - PDB of the protein structure
- "CT_TOX" - Clinical trial results

Parameters

- **featurizer** (ComplexFeaturizer *or str*) – the complex featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in
- **pocket** (*bool (default True)*) – If true, use only the binding pocket for featurization.

- **set_name** (*str (default 'core')*) – Name of dataset to download. 'refined', 'general', and 'core' are supported.

Returns

tasks, datasets, transformers –

tasks: list
Column names corresponding to machine learning target variables.

datasets: tuple
train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers: list
`deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

3.9.26 PPB Datasets

`load_ppb`(*featurizer:* Featurizer | *str = 'ECFP'*, *splitter:* Splitter | *str* | *None = 'scaffold'*, *transformers: List[TransformerGenerator* | *str] = ['normalization']*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load PPB datasets.

Parameters

- **featurizer** (Featurizer *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.27 QM7 Datasets

`load_qm7`(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = CoulombMatrix[max_atoms=23, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load QM7 dataset

QM7 is a subset of GDB-13 (a database of nearly 1 billion stable and synthetically accessible organic molecules) containing up to 7 heavy atoms C, N, O, and S. The 3D Cartesian coordinates of the most stable conformations and their atomization energies were determined using ab-initio density functional theory (PBE0/tier2 basis set). This dataset also provided Coulomb matrices as calculated in [Rupp et al. PRL, 2012]:

Stratified splitting is recommended for this dataset.

The data file (.mat format, we recommend using *scipy.io.loadmat* for python users to load this original data) contains five arrays:

- "X" - (7165 x 23 x 23), Coulomb matrices
- "T" - (7165), atomization energies (unit: kcal/mol)
- **"P" - (5 x 1433), cross-validation splits as used in [Montavon et al.**
  NIPS, 2012]
- "Z" - (7165 x 23), atomic charges
- **"R" - (7165 x 23 x 3), cartesian coordinate (unit: Bohr) of each atom in**
  the molecules

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Note: DeepChem 2.4.0 has turned on sanitization for this dataset by default. For the QM7 dataset, this means that calling this function will return 6838 compounds instead of 7160 in the source dataset file. This appears to be due to valence specification mismatches in the dataset that weren't caught in earlier more lax versions of RDKit. Note that this may subtly affect benchmarking results on this dataset.

References

3.9.28 QM8 Datasets

load_qm8(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = CoulombMatrix[max_atoms=26, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load QM8 dataset

QM8 is the dataset used in a study on modeling quantum mechanical calculations of electronic spectra and excited state energy of small molecules. Multiple methods, including time-dependent density functional theories (TDDFT) and second-order approximate coupled-cluster (CC2), are applied to a collection of molecules that include up to eight heavy atoms (also a subset of the GDB-17 database). In our collection, there are four excited state properties calculated by four different methods on 22 thousand samples:

S0 -> S1 transition energy E1 and the corresponding oscillator strength f1

S0 -> S2 transition energy E2 and the corresponding oscillator strength f2

E1, E2, f1, f2 are in atomic units. f1, f2 are in length representation

Random splitting is recommended for this dataset.

The source data contain:

- qm8.sdf: molecular structures
- qm8.sdf.csv: tables for molecular properties
- Column 1: Molecule ID (gdb9 index) mapping to the .sdf file
- Columns 2-5: RI-CC2/def2TZVP
- Columns 6-9: LR-TDPBE0/def2SVP
- Columns 10-13: LR-TDPBE0/def2TZVP
- Columns 14-17: LR-TDCAM-B3LYP/def2TZVP

Parameters

- **featurizer** (`Featurizer` *or* `str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter` *or* `str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

Note: DeepChem 2.4.0 has turned on sanitization for this dataset by default. For the QM8 dataset, this means that calling this function will return 21747 compounds instead of 21786 in the source dataset file. This appears to be due to valence specification mismatches in the dataset that weren't caught in earlier more lax versions of RDKit. Note that this may subtly affect benchmarking results on this dataset.

References

3.9.29 QM9 Datasets

`load_qm9`(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = CoulombMatrix[max_atoms=29, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load QM9 dataset

QM9 is a comprehensive dataset that provides geometric, energetic, electronic and thermodynamic properties for a subset of GDB-17 database, comprising 134 thousand stable organic molecules with up to 9 heavy atoms. All molecules are modeled using density functional theory (B3LYP/6-31G(2df,p) based DFT).

Random splitting is recommended for this dataset.

The source data contain:

- qm9.sdf: molecular structures
- qm9.sdf.csv: tables for molecular properties
- "mol_id" - Molecule ID (gdb9 index) mapping to the .sdf file
- "A" - Rotational constant (unit: GHz)
- "B" - Rotational constant (unit: GHz)
- "C" - Rotational constant (unit: GHz)
- "mu" - Dipole moment (unit: D)
- "alpha" - Isotropic polarizability (unit: Bohr^3)
- "homo" - Highest occupied molecular orbital energy (unit: Hartree)
- "lumo" - Lowest unoccupied molecular orbital energy (unit: Hartree)
- "gap" - Gap between HOMO and LUMO (unit: Hartree)
- "r2" - Electronic spatial extent (unit: Bohr^2)
- "zpve" - Zero point vibrational energy (unit: Hartree)
- "u0" - Internal energy at 0K (unit: Hartree)
- "u298" - Internal energy at 298.15K (unit: Hartree)
- "h298" - Enthalpy at 298.15K (unit: Hartree)
- "g298" - Free energy at 298.15K (unit: Hartree)
- "cv" - Heat capavity at 298.15K (unit: cal/(mol*K))
- "u0_atom" - Atomization energy at 0K (unit: kcal/mol)

- "u298_atom" - Atomization energy at 298.15K (unit: kcal/mol)
- "h298_atom" - Atomization enthalpy at 298.15K (unit: kcal/mol)
- "g298_atom" - Atomization free energy at 298.15K (unit: kcal/mol)

"u0_atom" ~ "g298_atom" (used in MoleculeNet) are calculated from the differences between "u0" ~ "g298" and sum of reference energies of all atoms in the molecules, as given in https://figshare.com/articles/Atomref%3A_Reference_thermochemical_energies_of_H%2C_C%2C_N%2C_O%2C_F_atoms./1057643

Parameters

- **`featurizer`** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **`splitter`** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **`transformers`** (`list of TransformerGenerators or strings`) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **`reload`** (`bool`) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **`data_dir`** (`str`) – a directory to save the raw data in
- **`save_dir`** (`str`) – a directory to save the dataset in

Note: DeepChem 2.4.0 has turned on sanitization for this dataset by default. For the QM9 dataset, this means that calling this function will return 132480 compounds instead of 133885 in the source dataset file. This appears to be due to valence specification mismatches in the dataset that weren't caught in earlier more lax versions of RDKit. Note that this may subtly affect benchmarking results on this dataset.

References

3.9.30 SAMPL Datasets

`load_sampl`(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load SAMPL(FreeSolv) dataset

The Free Solvation Database, FreeSolv(SAMPL), provides experimental and calculated hydration free energy of small molecules in water. The calculated values are derived from alchemical free energy calculations using molecular dynamics simulations. The experimental values are included in the benchmark collection.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "iupac" - IUPAC name of the compound
- "smiles" - SMILES representation of the molecular structure
- **"expt" - Measured solvation energy (unit: kcal/mol) of the compound,**
  used as label

- "calc" - Calculated solvation energy (unit: kcal/mol) of the compound

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.31 SIDER Datasets

load_sider(*featurizer: Featurizer | str = 'ECFP'*, *splitter: Splitter | str | None = 'scaffold'*, *transformers: List[TransformerGenerator | str] = ['balancing']*, *reload: bool = True*, *data_dir: str | None = None*, *save_dir: str | None = None*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load SIDER dataset

The Side Effect Resource (SIDER) is a database of marketed drugs and adverse drug reactions (ADR). The version of the SIDER dataset in DeepChem has grouped drug side effects into 27 system organ classes following MedDRA classifications measured for 1427 approved drugs.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "smiles": SMILES representation of the molecular structure
- **"Hepatobiliary disorders" ~ "Injury, poisoning and procedural**
  complications": Recorded side effects for the drug. Please refer to http://sideeffects.embl.de/se/?page=98 for details on ADRs.

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.

- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.32 Thermosol Datasets

load_thermosol(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = [], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Loads the thermodynamic solubility datasets.

Parameters

- **featurizer** (*Featurizer or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (*Splitter or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

3.9.33 Tox21 Datasets

load_tox21(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, tasks: List[str] = ['NR-AR', 'NR-AR-LBD', 'NR-AhR', 'NR-Aromatase', 'NR-ER', 'NR-ER-LBD', 'NR-PPAR-gamma', 'SR-ARE', 'SR-ATAD5', 'SR-HSE', 'SR-MMP', 'SR-p53'], **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Tox21 dataset

The "Toxicology in the 21st Century" (Tox21) initiative created a public database measuring toxicity of compounds, which has been used in the 2014 Tox21 Data Challenge. This dataset contains qualitative toxicity measurements for 8k compounds on 12 different targets, including nuclear receptors and stress response pathways.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "smiles" - SMILES representation of the molecular structure
- "NR-XXX" - Nuclear receptor signaling bioassays results
- "SR-XXX" - Stress response bioassays results

please refer to https://tripod.nih.gov/tox21/challenge/data.jsp for details.

Parameters

- **featurizer** (Featurizer *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in
- **tasks** (*List[str], (optional)*) – Specify the set of tasks to load. If no task is specified, then it loads
- **NR-AR** (*the default set of tasks which are*) –
- **NR-AR-LBD** –
- **NR-AhR** –
- **NR-Aromatase** –
- **NR-ER** –

:param : :param NR-ER-LBD: :param NR-PPAR-gamma: :param SR-ARE: :param SR-ATAD5: :param SR-HSE: :param SR-MMP: :param SR-p53.:

References

3.9.34 Toxcast Datasets

load_toxcast(*featurizer: Featurizer | str = 'ECFP', splitter: Splitter | str | None = 'scaffold', transformers: List[TransformerGenerator | str] = ['balancing'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Toxcast dataset

ToxCast is an extended data collection from the same initiative as Tox21, providing toxicology data for a large library of compounds based on in vitro high-throughput screening. The processed collection includes qualitative results of over 600 experiments on 8k compounds.

Random splitting is recommended for this dataset.

The raw data csv file contains columns below:

- "smiles": SMILES representation of the molecular structure
- **"ACEA_T47D_80hr_Negative" ~ "Tanguay_ZF_120hpf_YSE_up": Bioassays results.** Please refer to the section "high-throughput assay information" at https://www.epa.gov/chemical-research/toxicity-forecaster-toxcasttm-data for details.

Parameters

- **featurizer** (Featurizer *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.

- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in

References

3.9.35 USPTO Datasets

`load_uspto`(*featurizer:* Featurizer | *str = 'RxnFeaturizer'*, *splitter:* Splitter | *str* | *None = None*, *transformers: List[TransformerGenerator* | *str] = []*, *reload: bool = True*, *data_dir: str* | *None = None*, *save_dir: str* | *None = None*, *subset: str = 'MIT'*, *sep_reagent: bool = True*, ***kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load USPTO Datasets.

The USPTO dataset consists of over 1.8 Million organic chemical reactions extracted from US patents and patent applications. The dataset contains the reactions in the form of reaction SMILES, which have the general format: reactant>reagent>product.

Molnet provides ability to load subsets of the USPTO dataset namely MIT, STEREO and 50K. The MIT dataset contains around 479K reactions, curated by jin et al. The STEREO dataset contains around 1 Million Reactions, it does not have duplicates and the reactions include stereochemical information. The 50K dataset contatins 50,000 reactions and is the benchmark for retrosynthesis predictions. The reactions are additionally classified into 10 reaction classes. The canonicalized version of the dataset used by the loader is the same as that used by Somnath et. al.

The loader uses the SpecifiedSplitter to use the same splits as specified by Schwaller et. al and Dai et. al. Custom splitters could also be used. There is a toggle in the loader to skip the source/target transformation needed for seq2seq tasks. There is an additional toggle to load the dataset with the reagents and reactants separated or mixed. This alters the entries in source by replacing the '>' with '.' , effectively loading them as an unified SMILES string.

Parameters

- **featurizer** (Featurizer *or str*) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (Splitter *or str*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in

- **save_dir** (*str*) – a directory to save the dataset in
- **subset** (*str (default 'MIT')*) – Subset of dataset to download. 'FULL', 'MIT', 'STEREO', and '50K' are supported.
- **sep_reagent** (*bool (default True)*) – Toggle to load dataset with reactants and reagents either separated or mixed.
- **skip_transform** (*bool (default True)*) – Toggle to skip the source/target transformation.

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

References

3.9.36 UV Datasets

`load_uv`(*shard_size=2000*, *featurizer=None*, *split=None*, *reload=True*)

Load UV dataset; does not do train/test split

The UV dataset is an in-house dataset from Merck that was first introduced in the following paper: Ramsundar, Bharath, et al. "Is multitask deep learning practical for pharma?." Journal of chemical information and modeling 57.8 (2017): 2068-2076.

The UV dataset tests 10,000 of Merck's internal compounds on 190 absorption wavelengths between 210 and 400 nm. Unlike most of the other datasets featured in MoleculeNet, the UV collection does not have structures for the compounds tested since they were proprietary Merck compounds. However, the collection does feature pre-computed descriptors for these compounds.

Note that the original train/valid/test split from the source data was preserved here, so this function doesn't allow for alternate modes of splitting. Similarly, since the source data came pre-featurized, it is not possible to apply alternative featurizations.

Parameters

- **shard_size** (*int, optional*) – Size of the DiskDataset shards to write on disk
- **featurizer** (*optional*) – Ignored since featurization pre-computed
- **split** (*optional*) – Ignored since split pre-computed
- **reload** (*bool, optional*) – Whether to automatically re-load from disk

3.9.37 ZINC15 Datasets

load_zinc15(*featurizer: Featurizer | str = 'OneHot', splitter: Splitter | str | None = 'random', transformers: List[TransformerGenerator | str] = ['normalization'], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, dataset_size: str = '250K', dataset_dimension: str = '2D', tasks: List[str] = ['mwt', 'logp', 'reactive'], **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load zinc15.

ZINC15 is a dataset of over 230 million purchasable compounds for virtual screening of small molecules to identify structures that are likely to bind to drug targets. ZINC15 data is currently available in 2D (SMILES string) format.

MolNet provides subsets of 250K, 1M, and 10M "lead-like" compounds from ZINC15. The full dataset of 270M "goldilocks" compounds is also available. Compounds in ZINC15 are labeled by their molecular weight and LogP (solubility) values. Each compound also has information about how readily available (purchasable) it is and its reactivity. Lead-like compounds have molecular weight between 300 and 350 Daltons and LogP between -1 and 3.5. Goldilocks compounds are lead-like compounds with LogP values further restricted to between 2 and 3.

If *reload = True* and *data_dir* (*save_dir*) is specified, the loader will attempt to load the raw dataset (featurized dataset) from disk. Otherwise, the dataset will be downloaded from the DeepChem AWS bucket.

For more information on ZINC15, please see [1]_ and https://zinc15.docking.org/.

Parameters

- **featurizer** (`Featurizer or str`) – the featurizer to use for processing the data. Alternatively you can pass one of the names from dc.molnet.featurizers as a shortcut.
- **splitter** (`Splitter or str`) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings*) – the Transformers to apply to the data. Each one is specified by a TransformerGenerator or, as a shortcut, one of the names from dc.molnet.transformers.
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.
- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str*) – a directory to save the dataset in
- **size** (*str (default '250K')*) – Size of dataset to download. '250K', '1M', '10M', and '270M' are supported.
- **format** (*str (default '2D')*) – Format of data to download. 2D SMILES strings or 3D SDF files.
- **tasks** (List[str], (optional) default: *['molwt', 'logp', 'reactive']*) – Specify the set of tasks to load. If no task is specified, then it loads
- **molwt** (*the default set of tasks which are*) –
- **logp** –
- **reactive.** –

Returns

tasks, datasets, transformers –

tasks
[list] Column names corresponding to machine learning target variables.

datasets
[tuple] train, validation, test splits of data as `deepchem.data.datasets.Dataset` instances.

transformers
[list] `deepchem.trans.transformers.Transformer` instances applied to dataset.

Return type
tuple

Notes

The total ZINC dataset with SMILES strings contains hundreds of millions of compounds and is over 100GB! ZINC250K is recommended for experimentation. The full set of 270M goldilocks compounds is 23GB.

References

3.9.38 Platinum Adsorption Dataset

`load_Platinum_Adsorption`(*featurizer: ~deepchem.feat.base_classes.Featurizer | str = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: ~deepchem.splits.splitters.Splitter | str | None = 'random', transformers: ~typing.List[~deepchem.molnet.load_function.molnet_loader.TransformerGenerator | str] = [], reload: bool = True, data_dir: str | None = None, save_dir: str | None = None, **kwargs*) → Tuple[List[str], Tuple[*Dataset*, ...], List[*Transformer*]]

Load Platinum Adsorption Dataset

The dataset consist of diffrent configurations of Adsorbates (i.e N and NO) on Platinum surface represented as Lattice and their formation energy. There are 648 diffrent adsorbate configuration in this datasets represented as Pymatgen Structure objects.

1. **Pymatgen structure object with site_properties with following key value.**
   - **"SiteTypes", mentioning if it is a active site "A1" or spectator**
     site "S1".
   - "oss", diffrent occupational sites. For spectator sites make it -1.

Parameters

- **featurizer** (`Featurizer` *(default LCNNFeaturizer)*) – the featurizer to use for processing the data. Reccomended to use the LCNNFeaturiser.
- **splitter** (`Splitter` *(default RandomSplitter)*) – the splitter to use for splitting the data into training, validation, and test sets. Alternatively you can pass one of the names from dc.molnet.splitters as a shortcut. If this is None, all the data will be included in a single dataset.
- **transformers** (*list of TransformerGenerators or strings. the Transformers to*) – apply to the data and appropritate featuriser. Does'nt require any transformation for LCNN_featuriser
- **reload** (*bool*) – if True, the first call for a particular featurizer and splitter will cache the datasets to disk, and subsequent calls will reload the cached datasets.

- **data_dir** (*str*) – a directory to save the raw data in
- **save_dir** (*str, optional (default None)*) – a directory to save the dataset in

References

Examples

```
>>>
>> import deepchem as dc
>> tasks, datasets, transformers = load_Platinum_Adsorption(
>>    reload=True,
>>    data_dir=data_path,
>>    save_dir=data_path,
>>    featurizer_kwargs=feat_args)
>> train_dataset, val_dataset, test_dataset = datasets
```

3.10 Featurizers

DeepChem contains an extensive collection of featurizers. If you haven't run into this terminology before, a "featurizer" is chunk of code which transforms raw input data into a processed form suitable for machine learning. Machine learning methods often need data to be pre-chewed for them to process. Think of this like a mama penguin chewing up food so the baby penguin can digest it easily.

Now if you've watched a few introductory deep learning lectures, you might ask, why do we need something like a featurizer? Isn't part of the promise of deep learning that we can learn patterns directly from raw data?

Unfortunately it turns out that deep learning techniques need featurizers just like normal machine learning methods do. Arguably, they are less dependent on sophisticated featurizers and more capable of learning sophisticated patterns from simpler data. But nevertheless, deep learning systems can't simply chew up raw files. For this reason, `deepchem` provides an extensive collection of featurization methods which we will review on this page.

Contents

- *Molecule Featurizers*
  - *Graph Convolution Featurizers*
    - *ConvMolFeaturizer*
    - *WeaveFeaturizer*
    - *MolGanFeaturizer*
    - *MolGraphConvFeaturizer*
    - *PagtnMolGraphFeaturizer*
    - *DMPNNFeaturizer*
    - *GroverFeaturizer*
    - *RDKitConformerFeaturizer*
    - *MXMNetFeaturizer*

        * *Utilities*
    - *MACCSKeysFingerprint*
    - *MATFeaturizer*
    - *CircularFingerprint*
    - *PubChemFingerprint*
    - *Mol2VecFingerprint*
    - *RDKitDescriptors*
    - *MordredDescriptors*
    - *CoulombMatrix*
    - *CoulombMatrixEig*
    - *AtomCoordinates*
    - *BPSymmetryFunctionInput*
    - *SmilesToSeq*
    - *SmilesToImage*
    - *OneHotFeaturizer*
    - *SparseMatrixOneHotFeaturizer*
    - *RawFeaturizer*
    - *SNAPFeaturizer*
- *Molecular Complex Featurizers*
    - *RdkitGridFeaturizer*
    - *AtomicConvFeaturizer*
- *Inorganic Crystal Featurizers*
    - *MaterialCompositionFeaturizer*
        * *ElementPropertyFingerprint*
        * *ElemNetFeaturizer*
    - *MaterialStructureFeaturizer*
        * *SineCoulombMatrix*
        * *CGCNNFeaturizer*
    - *LCNNFeaturizer*
- *Biological Sequence Featurizers*
    - *SAMFeaturizer*
    - *BAMFeaturizer*
    - *CRAMFeaturizer*
- *Molecule Tokenizers*
    - *SmilesTokenizer*

  - *BasicSmilesTokenizer*
  - *HuggingFaceFeaturizer*
  - *GroverAtomVocabTokenizer*
  - *GroverBondVocabTokenizer*
- *Vocabulary Builders*
- *Sequence Featurizers*
  - *PFMFeaturizer*
- *Other Featurizers*
  - *BertFeaturizer*
  - *RobertaFeaturizer*
  - *RxnFeaturizer*
  - *BindingPocketFeaturizer*
  - *UserDefinedFeaturizer*
  - *DummyFeaturizer*
- *Base Featurizers (for develop)*
  - *Featurizer*
  - *MolecularFeaturizer*
  - *MaterialCompositionFeaturizer*
  - *MaterialStructureFeaturizer*
  - *ComplexFeaturizer*
  - *VocabularyBuilder*
  - *HuggingFaceVocabularyBuilder*

3.10.1 Molecule Featurizers

These featurizers work with datasets of molecules.

Graph Convolution Featurizers

We are simplifying our graph convolution models by a joint data representation (`GraphData`) in a future version of DeepChem, so we provide several featurizers.

`ConvMolFeaturizer` and `WeaveFeaturizer` are used with graph convolution models which inherited `KerasModel`. `ConvMolFeaturizer` is used with graph convolution models except `WeaveModel`. `WeaveFeaturizer` are only used with `WeaveModel`. On the other hand, `MolGraphConvFeaturizer` is used with graph convolution models which inherited `TorchModel`. `MolGanFeaturizer` will be used with MolGAN model, a GAN model for generation of small molecules.

ConvMolFeaturizer

class ConvMolFeaturizer(*master_atom: bool = False*, *use_chirality: bool = False*, *atom_properties: Iterable[str] = []*, *per_atom_fragmentation: bool = False*)

This class implements the featurization to implement Duvenaud graph convolutions.

Duvenaud graph convolutions [1]_ construct a vector of descriptors for each atom in a molecule. The featurizer computes that vector of local descriptors.

Examples

```
>>> import deepchem as dc
>>> smiles = ["C", "CCC"]
>>> featurizer=dc.feat.ConvMolFeaturizer(per_atom_fragmentation=False)
>>> f = featurizer.featurize(smiles)
>>> # Using ConvMolFeaturizer to create featurized fragments derived from molecules␣
↪of interest.
... # This is used only in the context of performing interpretation of models using␣
↪atomic
... # contributions (atom-based model interpretation)
... smiles = ["C", "CCC"]
>>> featurizer=dc.feat.ConvMolFeaturizer(per_atom_fragmentation=True)
>>> f = featurizer.featurize(smiles)
>>> len(f) # contains 2 lists with  featurized fragments from 2 mols
2
```

See also:

`Detailed`

References

Note: This class requires RDKit to be installed.

__init__(*master_atom: bool = False*, *use_chirality: bool = False*, *atom_properties: Iterable[str] = []*, *per_atom_fragmentation: bool = False*)

Parameters

- **master_atom** (*Boolean*) – if true create a fake atom with bonds to every other atom. the initialization is the mean of the other atom features in the molecule. This technique is briefly discussed in Neural Message Passing for Quantum Chemistry https://arxiv.org/pdf/1704.01212.pdf
- **use_chirality** (*Boolean*) – if true then make the resulting atom features aware of the chirality of the molecules in question
- **atom_properties** (*list of string or None*) – properties in the RDKit Mol object to use as additional atom-level features in the larger molecular feature. If None, then no atom-level properties are used. Properties should be in the RDKit mol object should be in the form atom XXXXXXXX NAME where XXXXXXXX is a zero-padded 8 digit number coresponding to the zero-indexed atom index of each atom and NAME is the name of the property provided in atom_properties. So "atom 00000000 sasa" would be the name of the

molecule level property in mol where the solvent accessible surface area of atom 0 would be stored.

- **per_atom_fragmentation** (*Boolean*) – If True, then multiple "atom-depleted" versions of each molecule will be created (using featurize() method). For each molecule, atoms are removed one at a time and the resulting molecule is featurized. The result is a list of ConvMol objects, one with each heavy atom removed. This is useful for subsequent model interpretation: finding atoms favorable/unfavorable for (modelled) activity. This option is typically used in combination with a FlatteningTransformer to split the lists into separate samples.

  Since ConvMol is an object and not a numpy array, need to set dtype to object.

featurize(*datapoints: Any | str | Iterable[Any] | Iterable[str]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Override parent: aim is to add handling atom-depleted molecules featurization

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

WeaveFeaturizer

class WeaveFeaturizer(*graph_distance: bool = True*, *explicit_H: bool = False*, *use_chirality: bool = False*, *max_pair_distance: int | None = None*)

This class implements the featurization to implement Weave convolutions.

Weave convolutions were introduced in [1]_. Unlike Duvenaud graph convolutions, weave convolutions require a quadratic matrix of interaction descriptors for each pair of atoms. These extra descriptors may provide for additional descriptive power but at the cost of a larger featurized dataset.

Examples

```
>>> import deepchem as dc
>>> mols = ["CCC"]
>>> featurizer = dc.feat.WeaveFeaturizer()
>>> features = featurizer.featurize(mols)
>>> type(features[0])
<class 'deepchem.feat.mol_graphs.WeaveMol'>
>>> features[0].get_num_atoms() # 3 atoms in compound
3
>>> features[0].get_num_features() # feature size
75
>>> type(features[0].get_atom_features())
<class 'numpy.ndarray'>
```

(continues on next page)

(continued from previous page)

```
>>> features[0].get_atom_features().shape
(3, 75)
>>> type(features[0].get_pair_features())
<class 'numpy.ndarray'>
>>> features[0].get_pair_features().shape
(9, 14)
```

References

Note: This class requires RDKit to be installed.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

__init__(*graph_distance: bool = True*, *explicit_H: bool = False*, *use_chirality: bool = False*, *max_pair_distance: int | None = None*)

Initialize this featurizer with set parameters.

Parameters

- **graph_distance** (*bool, (default True)*) – If True, use graph distance for distance features. Otherwise, use Euclidean distance. Note that this means that molecules that this featurizer is invoked on must have valid conformer information if this option is set.
- **explicit_H** (*bool, (default False)*) – If true, model hydrogens in the molecule.
- **use_chirality** (*bool, (default False)*) – If true, use chiral information in the featurization
- **max_pair_distance** (*Optional[int], (default None)*) – This value can be a positive integer or None. This parameter determines the maximum graph distance at which pair features are computed. For example, if *max_pair_distance==2*, then pair features are computed only for atoms at most graph distance 2 apart. If *max_pair_distance* is *None*, all pairs are considered (effectively infinite *max_pair_distance*)

MolGanFeaturizer

class MolGanFeaturizer(*max_atom_count: int = 9*, *kekulize: bool = True*, *bond_labels: List[Any] | None = None*, *atom_labels: List[int] | None = None*)

Featurizer for MolGAN de-novo molecular generation [1]_. The default representation is in form of GraphMatrix object. It is wrapper for two matrices containing atom and bond type information. The class also provides reverse capabilities.

Examples

```
>>> import deepchem as dc
>>> from rdkit import Chem
>>> rdkit_mol, smiles_mol = Chem.MolFromSmiles('CCC'), 'C1=CC=CC=C1'
>>> molecules = [rdkit_mol, smiles_mol]
>>> featurizer = dc.feat.MolGanFeaturizer()
>>> features = featurizer.featurize(molecules)
>>> len(features) # 2 molecules
2
>>> type(features[0])
<class 'deepchem.feat.molecule_featurizers.molgan_featurizer.GraphMatrix'>
>>> molecules = featurizer.defeaturize(features) # defeaturization
>>> type(molecules[0])
<class 'rdkit.Chem.rdchem.Mol'>
```

__init__(*max_atom_count: int = 9*, *kekulize: bool = True*, *bond_labels: List[Any] | None = None*, *atom_labels: List[int] | None = None*)

Parameters

- **max_atom_count** (*int, default 9*) – Maximum number of atoms used for creation of adjacency matrix. Molecules cannot have more atoms than this number Implicit hydrogens do not count.
- **kekulize** (*bool, default True*) – Should molecules be kekulized. Solves number of issues with defeaturization when used.
- **bond_labels** (*List[RDKitBond]*) – List of types of bond used for generation of adjacency matrix
- **atom_labels** (*List[int]*) – List of atomic numbers used for generation of node features

References

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

`defeaturize`(*graphs: GraphMatrix | Sequence[GraphMatrix]*, *log_every_n: int = 1000*) → ndarray

Calculates molecules from corresponding GraphMatrix objects.

Parameters

- **graphs** (*GraphMatrix / iterable*) – GraphMatrix object or corresponding iterable
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing RDKitMol objext.

Return type
np.ndarray

MolGraphConvFeaturizer

`class MolGraphConvFeaturizer`(*use_edges: bool = False*, *use_chirality: bool = False*, *use_partial_charge: bool = False*)

This class is a featurizer of general graph convolution networks for molecules.

The default node(atom) and edge(bond) representations are based on WeaveNet paper. If you want to use your own representations, you could use this class as a guide to define your original Featurizer. In many cases, it's enough to modify return values of *construct_atom_feature* or *construct_bond_feature*.

The default node representation are constructed by concatenating the following values, and the feature length is 30.

- Atom type: A one-hot vector of this atom, "C", "N", "O", "F", "P", "S", "Cl", "Br", "I", "other atoms".
- Formal charge: Integer electronic charge.
- Hybridization: A one-hot vector of "sp", "sp2", "sp3".
- Hydrogen bonding: A one-hot vector of whether this atom is a hydrogen bond donor or acceptor.
- Aromatic: A one-hot vector of whether the atom belongs to an aromatic ring.
- Degree: A one-hot vector of the degree (0-5) of this atom.
- Number of Hydrogens: A one-hot vector of the number of hydrogens (0-4) that this atom connected.
- Chirality: A one-hot vector of the chirality, "R" or "S". (Optional)
- Partial charge: Calculated partial charge. (Optional)

The default edge representation are constructed by concatenating the following values, and the feature length is 11.

- Bond type: A one-hot vector of the bond type, "single", "double", "triple", or "aromatic".
- Same ring: A one-hot vector of whether the atoms in the pair are in the same ring.
- Conjugated: A one-hot vector of whether this bond is conjugated or not.
- Stereo: A one-hot vector of the stereo configuration of a bond.

If you want to know more details about features, please check the paper [1]_ and utilities in deepchem.utils.molecule_feature_utils.py.

Examples

```
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> featurizer = MolGraphConvFeaturizer(use_edges=True)
>>> out = featurizer.featurize(smiles)
>>> type(out[0])
<class 'deepchem.feat.graph_data.GraphData'>
>>> out[0].num_node_features
30
>>> out[0].num_edge_features
11
```

References

Note: This class requires RDKit to be installed.

`__init__`(*use_edges: bool = False*, *use_chirality: bool = False*, *use_partial_charge: bool = False*)

Parameters

- **use_edges** (`bool, default False`) – Whether to use edge features or not.
- **use_chirality** (`bool, default False`) – Whether to use chirality information or not. If True, featurization becomes slow.
- **use_partial_charge** (`bool, default False`) – Whether to use partial charge data or not. If True, this featurizer computes gasteiger charges. Therefore, there is a possibility to fail to featurize for some molecules and featurization becomes slow.

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (`rdkit.Chem.rdchem.Mol / SMILES string / iterable`) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (`int, default 1000`) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

PagtnMolGraphFeaturizer

class PagtnMolGraphFeaturizer(*max_length=5*)

This class is a featuriser of PAGTN graph networks for molecules.

The featurization is based on PAGTN model. It is slightly more computationally intensive than default Graph Convolution Featuriser, but it builds a Molecular Graph connecting all atom pairs accounting for interactions of an atom with every other atom in the Molecule. According to the paper, interactions between two pairs of atom are dependent on the relative distance between them and and hence, the function needs to calculate the shortest path between them.

The default node representation is constructed by concatenating the following values, and the feature length is 94.

- Atom type: One hot encoding of the atom type. It consists of the most possible elements in a chemical compound.
- Formal charge: One hot encoding of formal charge of the atom.
- Degree: One hot encoding of the atom degree
- **Explicit Valence: One hot encoding of explicit valence of an atom. The supported possibilities**
  include `0 - 6`.
- **Implicit Valence: One hot encoding of implicit valence of an atom. The supported possibilities**
  include `0 - 5`.
- Aromaticity: Boolean representing if an atom is aromatic.

The default edge representation is constructed by concatenating the following values, and the feature length is 42. It builds a complete graph where each node is connected to every other node. The edge representations are calculated based on the shortest path between two nodes (choose any one if multiple exist). Each bond encountered in the shortest path is used to calculate edge features.

- Bond type: A one-hot vector of the bond type, "single", "double", "triple", or "aromatic".
- Conjugated: A one-hot vector of whether this bond is conjugated or not.
- Same ring: A one-hot vector of whether the atoms in the pair are in the same ring.
- Ring Size and Aromaticity: One hot encoding of atoms in pair based on ring size and aromaticity.
- Distance: One hot encoding of the distance between pair of atoms.

Examples

```
>>> from deepchem.feat import PagtnMolGraphFeaturizer
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> featurizer = PagtnMolGraphFeaturizer(max_length=5)
>>> out = featurizer.featurize(smiles)
>>> type(out[0])
<class 'deepchem.feat.graph_data.GraphData'>
>>> out[0].num_node_features
94
>>> out[0].num_edge_features
42
```

References

Note: This class requires RDKit to be installed.

`__init__`(*max_length=5*)

Parameters
`max_length` (*int*) – Maximum distance up to which shortest paths must be considered. Paths shorter than max_length will be padded and longer will be truncated, default to `5`.

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- `datapoints` (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- `log_every_n` (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

DMPNNFeaturizer

class `DMPNNFeaturizer`(*features_generators: List[str] | None = None*, *is_adding_hs: bool = False*, *use_original_atom_ranks: bool = False*)

This class is a featurizer for Directed Message Passing Neural Network (D-MPNN) implementation

The default node(atom) and edge(bond) representations are based on Analyzing Learned Molecular Representations for Property Prediction paper.

The default node representation are constructed by concatenating the following values, and the feature length is 133.

- Atomic num: A one-hot vector of this atom, in a range of first 100 atoms.
- Degree: A one-hot vector of the degree (0-5) of this atom.
- Formal charge: Integer electronic charge, -1, -2, 1, 2, 0.
- Chirality: A one-hot vector of the chirality tag (0-3) of this atom.
- Number of Hydrogens: A one-hot vector of the number of hydrogens (0-4) that this atom connected.
- Hybridization: A one-hot vector of "SP", "SP2", "SP3", "SP3D", "SP3D2".
- Aromatic: A one-hot vector of whether the atom belongs to an aromatic ring.
- Mass: Atomic mass * 0.01

The default edge representation are constructed by concatenating the following values, and the feature length is 14.

- Bond type: A one-hot vector of the bond type, "single", "double", "triple", or "aromatic".
- Same ring: A one-hot vector of whether the atoms in the pair are in the same ring.

- Conjugated: A one-hot vector of whether this bond is conjugated or not.
- Stereo: A one-hot vector of the stereo configuration (0-5) of a bond.

If you want to know more details about features, please check the paper [1]_ and utilities in deepchem.utils.molecule_feature_utils.py.

Examples

```
>>> smiles = ["C1=CC=CN=C1", "C1CCC1"]
>>> featurizer = DMPNNFeaturizer()
>>> out = featurizer.featurize(smiles)
>>> type(out[0])
<class 'deepchem.feat.graph_data.GraphData'>
>>> out[0].num_nodes
6
>>> out[0].num_node_features
133
>>> out[0].node_features.shape
(6, 133)
>>> out[0].num_edge_features
14
>>> out[0].num_edges
12
>>> out[0].edge_features.shape
(12, 14)
```

References

Note: This class requires RDKit to be installed.

`__init__`(*features_generators: List[str] | None = None*, *is_adding_hs: bool = False*, *use_original_atom_ranks: bool = False*)

Parameters

- **`features_generator`** (*`List[str], default None`*) – List of global feature generators to be used.
- **`is_adding_hs`** (*`bool, default False`*) – Whether to add Hs or not.
- **`use_original_atom_ranks`** (*`bool, default False`*) – Whether to use original atom mapping or canonical atom mapping

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **`datapoints`** (*`rdkit.Chem.rdchem.Mol / SMILES string / iterable`*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **`log_every_n`** (*`int, default 1000`*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

GroverFeaturizer

class GroverFeaturizer(*features_generator: MolecularFeaturizer | None = None*, *bond_drop_rate: float = 0.0*)

Featurizer for GROVER Model

The Grover Featurizer is used to compute features suitable for grover model. It accepts an rdkit molecule of type *rdkit.Chem.rdchem.Mol* or a SMILES string as input and computes the following sets of features:

1. a molecular graph from the input molecule
2. functional groups which are used **only** during pretraining
3. additional features which can **only** be used during finetuning

Parameters

- **additional_featurizer** (*dc.feat.Featurizer*) – Given a molecular dataset, it is possible to extract additional molecular features in order
- **can** (*to train and finetune from the existing pretrained model. The additional_featurizer*) –
- **molecule.** (*be used to generate additional features for the*) –

References

Examples

```
>>> import deepchem as dc
>>> from deepchem.feat import GroverFeaturizer
>>> feat = GroverFeaturizer(features_generator = dc.feat.CircularFingerprint())
>>> out = feat.featurize('CCC')
```

Note: This class requires RDKit to be installed.

__init__(*features_generator: MolecularFeaturizer | None = None*, *bond_drop_rate: float = 0.0*)

Parameters
use_original_atoms_order (*bool, default False*) – Whether to use original atom ordering or canonical ordering (default)

RDKitConformerFeaturizer

class RDKitConformerFeaturizer(*use_original_atoms_order=False*)

A featurizer that featurizes an RDKit mol object as a GraphData object with 3D coordinates. The 3D coordinates are represented in the node_pos_features attribute of the GraphData object of shape [num_atoms * num_conformers, 3].

The ETKDGv2 algorithm is used to generate 3D coordinates for the molecule. The RDKit source for this algorithm can be found in RDkit/Code/GraphMol/DistGeomHelpers/Embedder.cpp The documentation can be found here: https://rdkit.org/docs/source/rdkit.Chem.rdDistGeom.html#rdkit.Chem.rdDistGeom.ETKDGv2

This featurization requires RDKit.

Examples

```
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import␣
↪RDKitConformerFeaturizer
>>> featurizer = RDKitConformerFeaturizer()
>>> molecule = "CCO"
>>> conformer = featurizer.featurize(molecule)
>>> print (type(conformer[0]))
<class 'deepchem.feat.graph_data.GraphData'>
```

atom_to_feature_vector(*atom*)

Converts an RDKit atom object to a feature list of indices.

Parameters
`atom` (*Chem.rdchem.Atom*) – RDKit atom object.

Returns
List of feature indices for the given atom.

Return type
List[int]

bond_to_feature_vector(*bond*)

Converts an RDKit bond object to a feature list of indices.

Parameters
`bond` (*Chem.rdchem.Bond*) – RDKit bond object.

Returns
List of feature indices for the given bond.

Return type
List[int]

MXMNetFeaturizer

class MXMNetFeaturizer(*is_adding_hs: bool = False*)

This class is a featurizer for Multiplex Molecular Graph Neural Network (MXMNet) implementation.

The atomic numbers(indices) of atoms will be used later to generate randomly initialized trainable embeddings to be the input node embeddings.

This featurizer is based on Molecular Mechanics-Driven Graph Neural Network with Multiplex Graph for Molecular Structures.

Examples

```
>>> smiles = ["C1=CC=CN=C1", "C1CCC1"]
>>> featurizer = MXMNetFeaturizer()
>>> out = featurizer.featurize(smiles)
>>> type(out[0])
<class 'deepchem.feat.graph_data.GraphData'>
>>> out[0].num_nodes
6
>>> out[0].num_node_features
1
>>> out[0].node_features.shape
(6, 1)
>>> out[0].num_edges
12
```

Note: We are not explitly handling hydrogen atoms for now. We only support 'H', 'C', 'N', 'O' and 'F' atoms to be present in the smiles at this point for MXMNet Model.

__init__(*is_adding_hs: bool = False*)

Parameters
is_adding_hs (*bool, default False*) – Whether to add Hs or not.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

Utilities

Here are some constants that are used by the graph convolutional featurizers for molecules.

class GraphConvConstants

This class defines a collection of constants which are useful for graph convolutions on molecules.

possible_atom_list = ['C', 'N', 'O', 'S', 'F', 'P', 'Cl', 'Mg', 'Na', 'Br', 'Fe', 'Ca', 'Cu', 'Mc', 'Pd', 'Pb', 'K', 'I', 'Al', 'Ni', 'Mn']

Allowed Numbers of Hydrogens

possible_numH_list = [0, 1, 2, 3, 4]

Allowed Valences for Atoms

possible_valence_list = [0, 1, 2, 3, 4, 5, 6]

Allowed Formal Charges for Atoms

possible_formal_charge_list = [-3, -2, -1, 0, 1, 2, 3]

This is a placeholder for documentation. These will be replaced with corresponding values of the rdkit HybridizationType

possible_hybridization_list = ['SP', 'SP2', 'SP3', 'SP3D', 'SP3D2']

Allowed number of radical electrons.

possible_number_radical_e_list = [0, 1, 2]

Allowed types of Chirality

possible_chirality_list = ['R', 'S']

The set of all values allowed.

reference_lists = [['C', 'N', 'O', 'S', 'F', 'P', 'Cl', 'Mg', 'Na', 'Br', 'Fe', 'Ca', 'Cu', 'Mc', 'Pd', 'Pb', 'K', 'I', 'Al', 'Ni', 'Mn'], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5, 6], [-3, -2, -1, 0, 1, 2, 3], [0, 1, 2], ['SP', 'SP2', 'SP3', 'SP3D', 'SP3D2'], ['R', 'S']]

The number of different values that can be taken. See *get_intervals()*

intervals = [1, 6, 48, 384, 1536, 9216, 27648]

Possible stereochemistry. We use E-Z notation for stereochemistry https://en.wikipedia.org/wiki/E%E2%80%93Z_notation

possible_bond_stereo = ['STEREONONE', 'STEREOANY', 'STEREOZ', 'STEREOE']

Number of different bond types not counting stereochemistry.

bond_fdim_base = 6

__module__ = 'deepchem.feat.graph_features'

There are a number of helper methods used by the graph convolutional classes which we document here.

one_of_k_encoding(*x*, *allowable_set*)

Encodes elements of a provided set as integers.

Parameters

- **x** (*object*) – Must be present in *allowable_set*.
- **allowable_set** (*list*) – List of allowable quantities.

Example

```
>>> import deepchem as dc
>>> dc.feat.graph_features.one_of_k_encoding("a", ["a", "b", "c"])
[True, False, False]
```

Raises
`ValueError` –

`one_of_k_encoding_unk`(*x*, *allowable_set*)

Maps inputs not in the allowable set to the last element.

Unlike *one_of_k_encoding*, if *x* is not in *allowable_set*, this method pretends that *x* is the last element of *allowable_set*.

Parameters

- **`x`** (*`object`*) – Must be present in *allowable_set*.
- **`allowable_set`** (*`list`*) – List of allowable quantities.

Examples

```
>>> dc.feat.graph_features.one_of_k_encoding_unk("s", ["a", "b", "c"])
[False, False, True]
```

`get_intervals`(*l*)

For list of lists, gets the cumulative products of the lengths

Note that we add 1 to the lengths of all lists (to avoid an empty list propagating a 0).

Parameters
`l` (*`list of lists`*) – Returns the cumulative product of these lengths.

Examples

```
>>> dc.feat.graph_features.get_intervals([[1], [1, 2], [1, 2, 3]])
[1, 3, 12]
```

```
>>> dc.feat.graph_features.get_intervals([[1], [], [1, 2], [1, 2, 3]])
[1, 1, 3, 12]
```

`safe_index`(*l*, *e*)

Gets the index of e in l, providing an index of len(l) if not found

Parameters

- **`l`** (*`list`*) – List of values
- **`e`** (*`object`*) – Object to check whether *e* is in *l*

Examples

```
>>> dc.feat.graph_features.safe_index([1, 2, 3], 1)
0
>>> dc.feat.graph_features.safe_index([1, 2, 3], 7)
3
```

get_feature_list(*atom*)

Returns a list of possible features for this atom.

Parameters
atom (*RDKit.Chem.rdchem.Atom*) – Atom to get features for

Examples

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("C")
>>> atom = mol.GetAtoms()[0]
>>> features = dc.feat.graph_features.get_feature_list(atom)
>>> type(features)
<class 'list'>
>>> len(features)
6
```

Note: This method requires RDKit to be installed.

Returns
features – List of length 6. The i-th value in this list provides the index of the atom in the corresponding feature value list. The 6 feature values lists for this function are *[GraphConvConstants.possible_atom_list, GraphConvConstants.possible_numH_list, GraphConvConstants.possible_valence_list, GraphConvConstants.possible_formal_charge_list, GraphConvConstants.possible_num_radical_e_list]*.

Return type
list

features_to_id(*features*, *intervals*)

Convert list of features into index using spacings provided in intervals

Parameters

- **features** (*list*) – List of features as returned by *get_feature_list()*
- **intervals** (*list*) – List of intervals as returned by *get_intervals()*

Returns
id – The index in a feature vector given by the given set of features.

Return type
int

id_to_features(*id*, *intervals*)

Given an index in a feature vector, return the original set of features.

Parameters

- `id` (*int*) – The index in a feature vector given by the given set of features.
- `intervals` (*list*) – List of intervals as returned by *get_intervals()*

Returns
features – List of features as returned by *get_feature_list()*

Return type
list

`atom_to_id`(*atom*)

Return a unique id corresponding to the atom type

Parameters
`atom` (*RDKit.Chem.rdchem.Atom*) – Atom to convert to ids.

Returns
id – The index in a feature vector given by the given set of features.

Return type
int

This function helps compute distances between atoms from a given base atom.

`find_distance`(*a1: Any*, *num_atoms: int*, *bond_adj_list*, *max_distance=7*) → ndarray

Computes distances from provided atom.

Parameters

- `a1` (*RDKit atom*) – The source atom to compute distances from.
- `num_atoms` (*int*) – The total number of atoms.
- `bond_adj_list` (*list of lists*) – *bond_adj_list[i]* is a list of the atom indices that atom *i* shares a bond with. This list is symmetrical so if *j in bond_adj_list[i]* then *i in bond_adj_list[j]*.
- `max_distance` (*int, optional (default 7)*) – The max distance to search.

Returns
distances – Of shape *(num_atoms, max_distance)*. Provides a one-hot encoding of the distances. That is, *distances[i]* is a one-hot encoding of the distance from *a1* to atom *i*.

Return type
np.ndarray

This function is important and computes per-atom feature vectors used by graph convolutional featurizers.

`atom_features`(*atom*, *bool_id_feat=False*, *explicit_H=False*, *use_chirality=False*)

Helper method used to compute per-atom feature vectors.

Many different featurization methods compute per-atom features such as ConvMolFeaturizer, WeaveFeaturizer. This method computes such features.

Parameters

- `atom` (*RDKit.Chem.rdchem.Atom*) – Atom to compute features on.
- `bool_id_feat` (*bool, optional*) – Return an array of unique identifiers corresponding to atom type.
- `explicit_H` (*bool, optional*) – If true, model hydrogens explicitly
- `use_chirality` (*bool, optional*) – If true, use chirality information.

Returns
features – An array of per-atom features.

Return type
np.ndarray

Examples

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('CCC')
>>> atom = mol.GetAtoms()[0]
>>> features = dc.feat.graph_features.atom_features(atom)
>>> type(features)
<class 'numpy.ndarray'>
>>> features.shape
(75,)
```

This function computes the bond features used by graph convolutional featurizers.

`bond_features`(*bond*, *use_chirality=False*, *use_extended_chirality=False*)

Helper method used to compute bond feature vectors.

Many different featurization methods compute bond features such as WeaveFeaturizer. This method computes such features.

Parameters

- **bond** (*rdkit.Chem.rdchem.Bond*) – Bond to compute features on.
- **use_chirality** (*bool, optional*) – If true, use chirality information.
- **use_extended_chirality** (*bool, optional*) – If true, use chirality information with upto 6 different types.

Note: This method requires RDKit to be installed.

Returns

- **bond_feats** (*np.ndarray*) – Array of bond features. This is a 1-D array of length 6 if *use_chirality* is *False* else of length 10 with chirality encoded.
- **bond_feats** (*Sequence[Union[bool, int, float]]*) – List of bond features returned if *use_extended_chirality* is *True*.

Examples

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('CCC')
>>> bond = mol.GetBonds()[0]
>>> bond_features = dc.feat.graph_features.bond_features(bond)
>>> type(bond_features)
<class 'numpy.ndarray'>
>>> bond_features.shape
(6,)
```

Note: This method requires RDKit to be installed.

This function computes atom-atom features (for atom pairs which may not have bonds between them.)

`pair_features`(*mol: Any*, *bond_features_map: dict*, *bond_adj_list: List*, *bt_len: int = 6*, *graph_distance: bool = True*, *max_pair_distance: int | None = None*) → Tuple[ndarray, ndarray]

Helper method used to compute atom pair feature vectors.

Many different featurization methods compute atom pair features such as WeaveFeaturizer. Note that atom pair features could be for pairs of atoms which aren't necessarily bonded to one another.

Parameters

- **mol** (*RDKit Mol*) – Molecule to compute features on.
- **bond_features_map** (*dict*) – Dictionary that maps pairs of atom ids (say *(2, 3)* for a bond between atoms 2 and 3) to the features for the bond between them.
- **bond_adj_list** (*list of lists*) – *bond_adj_list[i]* is a list of the atom indices that atom *i* shares a bond with . This list is symmetrical so if *j in bond_adj_list[i]* then *i in bond_adj_list[j]*.
- **bt_len** (*int, optional (default 6)*) – The number of different bond types to consider.
- **graph_distance** (*bool, optional (default True)*) – If true, use graph distance between molecules. Else use euclidean distance. The specified *mol* must have a conformer. Atomic positions will be retrieved by calling *mol.getConformer(0)*.
- **max_pair_distance** (*Optional[int], (default None)*) – This value can be a positive integer or None. This parameter determines the maximum graph distance at which pair features are computed. For example, if *max_pair_distance==2*, then pair features are computed only for atoms at most graph distance 2 apart. If *max_pair_distance* is *None*, all pairs are considered (effectively infinite *max_pair_distance*)

Note: This method requires RDKit to be installed.

Returns

- **features** (*np.ndarray*) – Of shape *(N_edges, bt_len + max_distance + 1)*. This is the array of pairwise features for all atom pairs, where N_edges is the number of edges within max_pair_distance of one another in this molecules.
- **pair_edges** (*np.ndarray*) – Of shape *(2, num_pairs)* where *num_pairs* is the total number of pairs within *max_pair_distance* of one another.

MACCSKeysFingerprint

class MACCSKeysFingerprint

MACCS Keys Fingerprint.

The MACCS (Molecular ACCess System) keys are one of the most commonly used structural keys. Please confirm the details in [1]_, [2]_.

Examples

```
>>> import deepchem as dc
>>> smiles = 'CC(=O)OC1=CC=CC=C1C(=O)O'
>>> featurizer = dc.feat.MACCSKeysFingerprint()
>>> features = featurizer.featurize([smiles])
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(167,)
```

References

Note: This class requires RDKit to be installed.

__init__()

Initialize this featurizer.

MATFeaturizer

class MATFeaturizer

This class is a featurizer for the Molecule Attention Transformer [1]_. The returned value is a numpy array which consists of molecular graph descriptions:

- Node Features
- Adjacency Matrix
- Distance Matrix

References

Examples

```
>>> import deepchem as dc
>>> feat = dc.feat.MATFeaturizer()
>>> out = feat.featurize("CCC")
```

Note: This class requires RDKit to be installed.

`__init__()`

Parameters
`use_original_atoms_order` (*bool, default False*) – Whether to use original atom ordering or canonical ordering (default)

`construct_mol`(*mol: Any*) → Any

Processes an input RDKitMol further to be able to extract id-specific Conformers from it using mol.GetConformer().

Parameters
`mol` (*RDKitMol*) – RDKit Mol object.

Returns
mol – A processed RDKitMol object which is embedded, UFF Optimized and has Hydrogen atoms removed. If the former conditions are not met and there is a value error, then 2D Coordinates are computed instead.

Return type
RDKitMol

`atom_features`(*atom: Any*) → ndarray

Deepchem already contains an atom_features function, however we are defining a new one here due to the need to handle features specific to MAT. Since we need new features like Atom GetNeighbors and IsInRing, and the number of features required for MAT is a fraction of what the Deepchem atom_features function computes, we can speed up computation by defining a custom function.

Parameters
`atom` (*RDKitAtom*) – RDKit Atom object.

Returns
Numpy array containing atom features.

Return type
ndarray

`construct_node_features_matrix`(*mol: Any*) → ndarray

This function constructs a matrix of atom features for all atoms in a given molecule using the atom_features function.

Parameters
`mol` (*RDKitMol*) – RDKit Mol object.

Returns
Atom_features – Numpy array containing atom features.

Return type
ndarray

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

CircularFingerprint

class CircularFingerprint(*radius: int = 2*, *size: int = 2048*, *chiral: bool = False*, *bonds: bool = True*, *features: bool = False*, *sparse: bool = False*, *smiles: bool = False*, *is_counts_based: bool = False*)

Circular (Morgan) fingerprints.

Extended Connectivity Circular Fingerprints compute a bag-of-words style representation of a molecule by breaking it into local neighborhoods and hashing into a bit vector of the specified size. It is used specifically for structure-activity modelling. See [1]_ for more details.

References

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> from rdkit import Chem
>>> smiles = ['C1=CC=CC=C1']
>>> # Example 1: (size = 2048, radius = 4)
>>> featurizer = dc.feat.CircularFingerprint(size=2048, radius=4)
>>> features = featurizer.featurize(smiles)
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(2048,)
```

```
>>> # Example 2: (size = 2048, radius = 4, sparse = True, smiles = True)
>>> featurizer = dc.feat.CircularFingerprint(size=2048, radius=8,
...                                          sparse=True, smiles=True)
>>> features = featurizer.featurize(smiles)
>>> type(features[0]) # dict containing fingerprints
<class 'dict'>
```

__init__(*radius: int = 2*, *size: int = 2048*, *chiral: bool = False*, *bonds: bool = True*, *features: bool = False*, *sparse: bool = False*, *smiles: bool = False*, *is_counts_based: bool = False*)

Parameters

- **radius** (*int, optional (default 2)*) – Fingerprint radius.
- **size** (*int, optional (default 2048)*) – Length of generated bit vector.
- **chiral** (*bool, optional (default False)*) – Whether to consider chirality in fingerprint generation.
- **bonds** (*bool, optional (default True)*) – Whether to consider bond order in fingerprint generation.

- **features** (*bool, optional (default False)*) – Whether to use feature information instead of atom information; see RDKit docs for more info.
- **sparse** (*bool, optional (default False)*) – Whether to return a dict for each molecule containing the sparse fingerprint.
- **smiles** (*bool, optional (default False)*) – Whether to calculate SMILES strings for fragment IDs (only applicable when calculating sparse fingerprints).
- **is_counts_based** (*bool, optional (default False)*) – Whether to generates a counts-based fingerprint.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

PubChemFingerprint

class PubChemFingerprint

PubChem Fingerprint.

The PubChem fingerprint is a 881 bit structural key, which is used by PubChem for similarity searching. Please confirm the details in [1]_.

References

Note: This class requires RDKit and PubChemPy to be installed. PubChemPy use REST API to get the fingerprint, so you need the internet access.

Examples

```
>>> import deepchem as dc
>>> smiles = ['CCC']
>>> featurizer = dc.feat.PubChemFingerprint()
>>> features = featurizer.featurize(smiles)
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(881,)
```

__init__()

Initialize this featurizer.

Mol2VecFingerprint

class Mol2VecFingerprint(*pretrain_model_path: str | None = None*, *radius: int = 1*, *unseen: str = 'UNK'*)

Mol2Vec fingerprints.

This class convert molecules to vector representations by using Mol2Vec. Mol2Vec is an unsupervised machine learning approach to learn vector representations of molecular substructures and the algorithm is based on Word2Vec, which is one of the most popular technique to learn word embeddings using neural network in NLP. Please see the details from **[1]_**.

The Mol2Vec requires the pretrained model, so we use the model which is put on the mol2vec github repository **[2]_**. The default model was trained on 20 million compounds downloaded from ZINC using the following paramters.

- radius 1
- UNK to replace all identifiers that appear less than 4 times
- skip-gram and window size of 10
- embeddings size 300

References

Note: This class requires mol2vec to be installed.

Examples

```
>>> import deepchem as dc
>>> from rdkit import Chem
>>> smiles = ['CCC']
>>> featurizer = dc.feat.Mol2VecFingerprint()
>>> features = featurizer.featurize(smiles)
>>> type(features)
<class 'numpy.ndarray'>
>>> features[0].shape
(300,)
```

__init__(*pretrain_model_path: str | None = None*, *radius: int = 1*, *unseen: str = 'UNK'*)

Parameters

- **pretrain_file** (*str, optional*) – The path for pretrained model. If this value is None, we use the model which is put on github repository (https://github.com/samoturk/mol2vec/tree/master/examples/models). The model is trained on 20 million compounds downloaded from ZINC.
- **radius** (*int, optional (default 1)*) – The fingerprint radius. The default value was used to train the model which is put on github repository.

- **unseen** (*str, optional (default 'UNK')*) – The string to used to replace uncommon words/identifiers while training.

`sentences2vec`(*sentences: list*, *model*, *unseen=None*) → ndarray

Generate vectors for each sentence (list) in a list of sentences. Vector is simply a sum of vectors for individual words.

Parameters

- **sentences** (*list, array*) – List with sentences
- **model** (*word2vec.Word2Vec*) – Gensim word2vec model
- **unseen** (*None, str*) – Keyword for unseen words. If None, those words are skipped. https://stats.stackexchange.com/questions/163005/how-to-set-the-dictionary-for-text-analysis-using-neural-networks/163032#163032

Return type
np.array

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

RDKitDescriptors

class `RDKitDescriptors`(*descriptors: List[str] = []*, *is_normalized: bool = False*, *use_fragment: bool = True*, *ipc_avg: bool = True*, *use_bcut2d: bool = True*, *labels_only: bool = False*)

RDKit descriptors.

This class computes a list of chemical descriptors like molecular weight, number of valence electrons, maximum and minimum partial charge, etc using RDKit.

This class can also compute normalized descriptors, if required. (The implementation for normalization is based on *RDKit2DNormalized()* method in 'descriptastorus' library.)

When the *is_normalized* option is set as True, descriptor values are normalized across the sample by fitting a cumulative density function. CDFs were used as opposed to simpler scaling algorithms mainly because CDFs have the useful property that 'each value has the same meaning: the percentage of the population observed below the raw feature value.'

Warning: Currently, the normalizing cdf parameters are not available for BCUT2D descriptors. (BCUT2D_MWHI, BCUT2D_MWLOW, BCUT2D_CHGHI, BCUT2D_CHGLO, BCUT2D_LOGPHI, BCUT2D_LOGPLOW, BCUT2D_MRHI, BCUT2D_MRLOW)

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> smiles = ['CC(=O)OC1=CC=CC=C1C(=O)O']
>>> featurizer = dc.feat.RDKitDescriptors()
>>> features = featurizer.featurize(smiles)
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(210,)
```

`__init__`(*descriptors: List[str] = [], is_normalized: bool = False, use_fragment: bool = True, ipc_avg: bool = True, use_bcut2d: bool = True, labels_only: bool = False*)

Initialize this featurizer.

Parameters

- **`descriptors`** (*`List[str] (default None)`*) – List of RDKit descriptors to compute properties. When None, computes values
- **`arguments.`** (*`for descriptors which are chosen based on options set in other`*) –
- **`use_fragment`** (*`bool, optional (default True)`*) – If True, the return value includes the fragment binary descriptors like 'fr_XXX'.
- **`ipc_avg`** (*`bool, optional (default True)`*) – If True, the IPC descriptor calculates with avg=True option. Please see this issue: https://github.com/rdkit/rdkit/issues/1527.
- **`is_normalized`** (*`bool, optional (default False)`*) – If True, the return value contains normalized features.
- **`use_bcut2d`** (*`bool, optional (default True)`*) – If True, the return value includes the descriptors like 'BCUT2D_XXX'.
- **`labels_only`** (*`bool, optional (default False)`*) – Returns only the presence or absence of a group.

Notes

- **If both *labels_only* and *is_normalized* are True, then *is_normalized* takes**
  precendence and *labels_only* will not be applied.

`featurize`(*datapoints, log_every_n=1000, **kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **`datapoints`** (*`rdkit.Chem.rdchem.Mol / SMILES string / iterable`*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **`log_every_n`** (*`int, default 1000`*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

MordredDescriptors

class MordredDescriptors(*ignore_3D: bool = True*)

Mordred descriptors.

This class computes a list of chemical descriptors using Mordred. Please see the details about all descriptors from [1]_, [2]_.

descriptors

List of Mordred descriptor names used in this class.

Type

List[str]

References

Note: This class requires Mordred to be installed.

Examples

```
>>> import deepchem as dc
>>> smiles = ['CC(=O)OC1=CC=CC=C1C(=O)O']
>>> featurizer = dc.feat.MordredDescriptors(ignore_3D=True)
>>> features = featurizer.featurize(smiles)
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(1613,)
```

__init__(*ignore_3D: bool = True*)

Parameters

ignore_3D (*bool, optional (default True)*) – Whether to use 3D information or not.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

CoulombMatrix

class CoulombMatrix(*max_atoms: int*, *remove_hydrogens: bool = False*, *randomize: bool = False*, *upper_tri: bool = False*, *n_samples: int = 1*, *seed: int | None = None*)

Calculate Coulomb matrices for molecules.

Coulomb matrices provide a representation of the electronic structure of a molecule. For a molecule with N atoms, the Coulomb matrix is a $N X N$ matrix where each element gives the strength of the electrostatic interaction between two atoms. The method is described in more detail in [1]_.

Examples

```
>>> import deepchem as dc
>>> featurizers = dc.feat.CoulombMatrix(max_atoms=23)
>>> input_file = 'deepchem/feat/tests/data/water.sdf' # really backed by water.sdf.
↪csv
>>> tasks = ["atomization_energy"]
>>> loader = dc.data.SDFLoader(tasks, featurizer=featurizers)
>>> dataset = loader.create_dataset(input_file)
```

References

Note: This class requires RDKit to be installed.

__init__(*max_atoms: int*, *remove_hydrogens: bool = False*, *randomize: bool = False*, *upper_tri: bool = False*, *n_samples: int = 1*, *seed: int | None = None*)

Initialize this featurizer.

Parameters

- **max_atoms** (*int*) – The maximum number of atoms expected for molecules this featurizer will process.
- **remove_hydrogens** (*bool, optional (default False)*) – If True, remove hydrogens before processing them.
- **randomize** (*bool, optional (default False)*) – If True, use method *randomize_coulomb_matrices* to randomize Coulomb matrices.
- **upper_tri** (*bool, optional (default False)*) – Generate only upper triangle part of Coulomb matrices.
- **n_samples** (*int, optional (default 1)*) – If *randomize* is set to True, the number of random samples to draw.
- **seed** (*int, optional (default None)*) – Random seed to use.

coulomb_matrix(*mol: Any*) → ndarray

Generate Coulomb matrices for each conformer of the given molecule.

Parameters
mol (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
The coulomb matrices of the given molecule

Return type
np.ndarray

randomize_coulomb_matrix(*m: ndarray*) → List[ndarray]

Randomize a Coulomb matrix as decribed in [1]_:

1. Compute row norms for M in a vector row_norms.
2. **Sample a zero-mean unit-variance noise vector e with dimension**
   equal to row_norms.
3. **Permute the rows and columns of M with the permutation that**
   sorts row_norms + e.

Parameters
m (*np.ndarray*) – Coulomb matrix.

Returns
List of the random coulomb matrix

Return type
List[np.ndarray]

References

static **get_interatomic_distances**(*conf: Any*) → ndarray

Get interatomic distances for atoms in a molecular conformer.

Parameters
conf (*rdkit.Chem.rdchem.Conformer*) – Molecule conformer.

Returns
The distances matrix for all atoms in a molecule

Return type
np.ndarray

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

CoulombMatrixEig

class CoulombMatrixEig(*max_atoms: int*, *remove_hydrogens: bool = False*, *randomize: bool = False*, *n_samples: int = 1*, *seed: int | None = None*)

Calculate the eigenvalues of Coulomb matrices for molecules.

This featurizer computes the eigenvalues of the Coulomb matrices for provided molecules. Coulomb matrices are described in [1]_.

Examples

```
>>> import deepchem as dc
>>> featurizers = dc.feat.CoulombMatrixEig(max_atoms=23)
>>> input_file = 'deepchem/feat/tests/data/water.sdf' # really backed by water.sdf.
↪csv
>>> tasks = ["atomization_energy"]
>>> loader = dc.data.SDFLoader(tasks, featurizer=featurizers)
>>> dataset = loader.create_dataset(input_file)
```

References

__init__(*max_atoms: int*, *remove_hydrogens: bool = False*, *randomize: bool = False*, *n_samples: int = 1*, *seed: int | None = None*)

Initialize this featurizer.

Parameters

- **max_atoms** (*int*) – The maximum number of atoms expected for molecules this featurizer will process.
- **remove_hydrogens** (*bool, optional (default False)*) – If True, remove hydrogens before processing them.
- **randomize** (*bool, optional (default False)*) – If True, use method *randomize_coulomb_matrices* to randomize Coulomb matrices.
- **n_samples** (*int, optional (default 1)*) – If *randomize* is set to True, the number of random samples to draw.
- **seed** (*int, optional (default None)*) – Random seed to use.

coulomb_matrix(*mol: Any*) → ndarray

Generate Coulomb matrices for each conformer of the given molecule.

Parameters
mol (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
The coulomb matrices of the given molecule

Return type
np.ndarray

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

static get_interatomic_distances(*conf: Any*) → ndarray

Get interatomic distances for atoms in a molecular conformer.

Parameters
conf (*rdkit.Chem.rdchem.Conformer*) – Molecule conformer.

Returns
The distances matrix for all atoms in a molecule

Return type
np.ndarray

randomize_coulomb_matrix(*m: ndarray*) → List[ndarray]

Randomize a Coulomb matrix as decribed in [1]_:

1. Compute row norms for M in a vector row_norms.
2. **Sample a zero-mean unit-variance noise vector e with dimension**
   equal to row_norms.
3. **Permute the rows and columns of M with the permutation that**
   sorts row_norms + e.

Parameters
m (*np.ndarray*) – Coulomb matrix.

Returns
List of the random coulomb matrix

Return type
List[np.ndarray]

References

AtomCoordinates

class AtomicCoordinates(*use_bohr: bool = False*)

Calculate atomic coordinates.

Examples

```
>>> import deepchem as dc
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('C1C=CC=CC=1')
>>> n_atoms = len(mol.GetAtoms())
>>> n_atoms
6
>>> featurizer = dc.feat.AtomicCoordinates(use_bohr=False)
>>> features = featurizer.featurize([mol])
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape # (n_atoms, 3)
(6, 3)
```

Note: This class requires RDKit to be installed.

`__init__`(*use_bohr: bool = False*)

Parameters
`use_bohr` (*bool, optional (default False)*) – Whether to use bohr or angstrom as a coordinate unit.

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray
Calculate features for molecules.

Parameters

- `datapoints` (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- `log_every_n` (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

BPSymmetryFunctionInput

class `BPSymmetryFunctionInput`(*max_atoms: int*)
Calculate symmetry function for each atom in the molecules

This method is described in [1]_.

Examples

```
>>> import deepchem as dc
>>> smiles = ['C1C=CC=CC=1']
>>> featurizer = dc.feat.BPSymmetryFunctionInput(max_atoms=10)
>>> features = featurizer.featurize(smiles)
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape  # (max_atoms, 4)
(10, 4)
```

References

Note: This class requires RDKit to be installed.

`__init__`(*max_atoms: int*)

Initialize this featurizer.

Parameters

`max_atoms` (*`int`*) – The maximum number of atoms expected for molecules this featurizer will process.

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **`datapoints`** (*`rdkit.Chem.rdchem.Mol / SMILES string / iterable`*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **`log_every_n`** (*`int, default 1000`*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

SmilesToSeq

class `SmilesToSeq`(*char_to_idx: Dict[str, int]*, *max_len: int = 250*, *pad_len: int = 10*)

SmilesToSeq Featurizer takes a SMILES string, and turns it into a sequence. Details taken from [1]_.

SMILES strings smaller than a specified max length (max_len) are padded using the PAD token while those larger than the max length are not considered. Based on the paper, there is also the option to add extra padding (pad_len) on both sides of the string after length normalization. Using a character to index (char_to_idx) mapping, the SMILES characters are turned into indices and the resulting sequence of indices serves as the input for an embedding layer.

References

Note: This class requires RDKit to be installed.

`__init__`(*char_to_idx: Dict[str, int]*, *max_len: int = 250*, *pad_len: int = 10*)

Initialize this class.

Parameters

- **char_to_idx** (*Dict*) – Dictionary containing character to index mappings for unique characters
- **max_len** (*int, default 250*) – Maximum allowed length of the SMILES string.
- **pad_len** (*int, default 10*) – Amount of padding to add on either side of the SMILES seq

`to_seq`(*smile: List[str]*) → ndarray

Turns list of smiles characters into array of indices

`remove_pad`(*characters: List[str]*) → List[str]

Removes PAD_TOKEN from the character list.

`smiles_from_seq`(*seq: List[int]*) → str

Reconstructs SMILES string from sequence.

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

SmilesToImage

class SmilesToImage(*img_size: int = 80*, *res: float = 0.5*, *max_len: int = 250*, *img_spec: str = 'std'*)

Convert SMILES string to an image.

SmilesToImage Featurizer takes a SMILES string, and turns it into an image. Details taken from [1]_.

The default size of for the image is 80 x 80. Two image modes are currently supported - std & engd. std is the gray scale specification, with atomic numbers as pixel values for atom positions and a constant value of 2 for bond positions. engd is a 4-channel specification, which uses atom properties like hybridization, valency, charges in addition to atomic number. Bond type is also used for the bonds.

The coordinates of all atoms are computed, and lines are drawn between atoms to indicate bonds. For the respective channels, the atom and bond positions are set to the property values as mentioned in the paper.

Examples

```
>>> import deepchem as dc
>>> smiles = ['CC(=O)OC1=CC=CC=C1C(=O)O']
>>> featurizer = dc.feat.SmilesToImage(img_size=80, img_spec='std')
>>> images = featurizer.featurize(smiles)
>>> type (images[0])
<class 'numpy.ndarray'>
>>> images[0].shape # (img_size, img_size, 1)
(80, 80, 1)
```

References

Note: This class requires RDKit to be installed.

`__init__`(*img_size: int = 80*, *res: float = 0.5*, *max_len: int = 250*, *img_spec: str = 'std'*)

Parameters

- **`img_size`** (*`int, default 80`*) – Size of the image tensor
- **`res`** (*`float, default 0.5`*) – Displays the resolution of each pixel in Angstrom
- **`max_len`** (*`int, default 250`*) – Maximum allowed length of SMILES string
- **`img_spec`** (*`str, default std`*) – Indicates the channel organization of the image tensor

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **`datapoints`** (*`rdkit.Chem.rdchem.Mol / SMILES string / iterable`*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **`log_every_n`** (*`int, default 1000`*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

OneHotFeaturizer

`class OneHotFeaturizer`(*charset: List[str] = ['#', ')', '(', '+', '-', '/', '1', '3', '2', '5', '4', '7', '6', '8', '=', '@', 'C', 'B', 'F', 'I', 'H', 'O', 'N', 'S', '[', ']', '\\', 'c', 'l', 'o', 'n', 'p', 's', 'r']*, *max_length: int | None = 100*)

Encodes any arbitrary string or molecule as a one-hot array.

This featurizer encodes the characters within any given string as a one-hot array. It also works with RDKit molecules: it can convert RDKit molecules to SMILES strings and then one-hot encode the characters in said strings.

Standalone Usage:

```
>>> import deepchem as dc
>>> featurizer = dc.feat.OneHotFeaturizer()
>>> smiles = ['CCC']
>>> encodings = featurizer.featurize(smiles)
>>> type(encodings[0])
<class 'numpy.ndarray'>
>>> encodings[0].shape
(100, 35)
>>> featurizer.untransform(encodings[0])
'CCC'
```

Note: This class needs RDKit to be installed in order to accept RDKit molecules as inputs.

It does not need RDKit to be installed to work with arbitrary strings.

`__init__`(*charset: List[str] = ['#', ')', '(', '+', '-', '/', '1', '3', '2', '5', '4', '7', '6', '8', '=', '@', 'C', 'B', 'F', 'I', 'H', 'O', 'N', 'S', '[', ']', '\\', 'c', 'l', 'o', 'n', 'p', 's', 'r'], max_length: int | None = 100*)

Initialize featurizer.

Parameters

- **`charset`** (*`List[str] (default ZINC_CHARSET)`*) – A list of strings, where each string is length 1 and unique.
- **`max_length`** (*`Optional[int], optional (default 100)`*) – The max length for string. If the length of string is shorter than max_length, the string is padded using space.
- **`None`** (*`If max_length is`*) –
- **`length`** (*`no padding is performed and arbitrary`*) –
- **`allowed.`** (*`strings are`*) –

`featurize`(*datapoints: Iterable[Any], log_every_n: int = 1000, **kwargs*) → ndarray

Featurize strings or mols.

Parameters

- **`datapoints`** (*`list`*) – A list of either strings (str or numpy.str_) or RDKit molecules.
- **`log_every_n`** (*`int, optional (default 1000)`*) – How many elements are featurized every time a featurization is logged.

`pad_smile`(*smiles: str*) → str

Pad SMILES string to *self.pad_length*

Parameters
`smiles` (*`str`*) – The SMILES string to be padded.

Returns
SMILES string space padded to self.pad_length

Return type
str

`pad_string`(*string: str*) → str

Pad string to *self.pad_length*

Parameters
`string` (*str*) – The string to be padded.

Returns
String space padded to self.pad_length

Return type
str

`untransform`(*one_hot_vectors: ndarray*) → str

Convert from one hot representation back to original string

Parameters
`one_hot_vectors` (*np.ndarray*) – An array of one hot encoded features.

Returns
Original string for an one hot encoded array.

Return type
str

SparseMatrixOneHotFeaturizer

`class SparseMatrixOneHotFeaturizer`(*charset: List[str] = ['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', 'X', 'Z', 'B', 'U', 'O']*)

Encodes any arbitrary string as a one-hot array.

This featurizer uses the sklearn OneHotEncoder to create sparse matrix representation of a one-hot array of any string. It is expected to be used in large datasets that produces memory overload using standard featurizer such as OneHotFeaturizer. For example: SwissprotDataset

Examples

```
>>> import deepchem as dc
>>> featurizer = dc.feat.SparseMatrixOneHotFeaturizer()
>>> sequence = "MMMQLA"
>>> encodings = featurizer.featurize([sequence])
>>> encodings[0].shape
(6, 25)
```

`__init__`(*charset: List[str] = ['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', 'X', 'Z', 'B', 'U', 'O']*)

Initialize featurizer.

Parameters
`charset` (*List[str] (default code)*) – A list of strings, where each string is length 1 and unique.

`featurize`(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Featurize strings.

Parameters

- `datapoints` (*list*) – A list of either strings (str or numpy.str_)
- `log_every_n` (*int, optional (default 1000)*) – How many elements are featurized every time a featurization is logged.

untransform(*one_hot_vectors: spmatrix*) → str

Convert from one hot representation back to original string

Parameters
one_hot_vectors (*np.ndarray*) – An array of one hot encoded features.

Returns
Original string for an one hot encoded array.

Return type
str

RawFeaturizer

class RawFeaturizer(*smiles: bool = False*)

Encodes a molecule as a SMILES string or RDKit mol.

This featurizer can be useful when you're trying to transform a large collection of RDKit mol objects as Smiles strings, or alternatively as a "no-op" featurizer in your molecular pipeline.

Note: This class requires RDKit to be installed.

__init__(*smiles: bool = False*)

Initialize this featurizer.

Parameters
smiles (*bool, optional (default False)*) – If True, encode this molecule as a SMILES string. Else as a RDKit mol.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

SNAPFeaturizer

class SNAPFeaturizer(*use_original_atoms_order=False*)

This featurizer is based on the SNAP featurizer used in the paper [1].

Example

```
>>> smiles = ["CC(=O)C"]
>>> featurizer = SNAPFeaturizer()
>>> print(featurizer.featurize(smiles))
[GraphData(node_features=[4, 2], edge_index=[2, 6], edge_features=[6, 2])]
```

References

`__init__`(*use_original_atoms_order=False*)

Parameters

`use_original_atoms_order` (*`bool, default False`*) – Whether to use original atom ordering or canonical ordering (default)

`featurize`(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **`datapoints`** (*`rdkit.Chem.rdchem.Mol / SMILES string / iterable`*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **`log_every_n`** (*`int, default 1000`*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

3.10.2 Molecular Complex Featurizers

These featurizers work with three dimensional molecular complexes.

RdkitGridFeaturizer

class `RdkitGridFeaturizer`(*nb_rotations=0*, *feature_types=None*, *ecfp_degree=2*, *ecfp_power=3*, *splif_power=3*, *box_width=16.0*, *voxel_width=1.0*, *flatten=False*, *verbose=True*, *sanitize=False*, ***kwargs*)

Featurizes protein-ligand complex using flat features or a 3D grid (in which each voxel is described with a vector of features).

`__init__`(*nb_rotations=0*, *feature_types=None*, *ecfp_degree=2*, *ecfp_power=3*, *splif_power=3*, *box_width=16.0*, *voxel_width=1.0*, *flatten=False*, *verbose=True*, *sanitize=False*, ***kwargs*)

Parameters

- **`nb_rotations`** (*`int, optional (default 0)`*) – Number of additional random rotations of a complex to generate.
- **`feature_types`** (*`list, optional (default ['ecfp'])`*) –

Types of features to calculate. Available types are
flat features -> 'ecfp_ligand', 'ecfp_hashed', 'splif_hashed', 'hbond_count' voxel features -> 'ecfp', 'splif', 'sybyl', 'salt_bridge', 'charge', 'hbond', 'pi_stack, 'cation_pi'

There are also 3 predefined sets of features
'flat_combined', 'voxel_combined', and 'all_combined'.

Calculated features are concatenated and their order is preserved (features in predefined sets are in alphabetical order).

- **ecfp_degree** (*int, optional (default 2)*) – ECFP radius.
- **ecfp_power** (*int, optional (default 3)*) – Number of bits to store ECFP features (resulting vector will be 2^ecfp_power long)
- **splif_power** (*int, optional (default 3)*) – Number of bits to store SPLIF features (resulting vector will be 2^splif_power long)
- **box_width** (*float, optional (default 16.0)*) – Size of a box in which voxel features are calculated. Box is centered on a ligand centroid.
- **voxel_width** (*float, optional (default 1.0)*) – Size of a 3D voxel in a grid.
- **flatten** (*bool, optional (defaul False)*) – Indicate whether calculated features should be flattened. Output is always flattened if flat features are specified in feature_types.
- **verbose** (*bool, optional (defaul True)*) – Verbolity for logging
- **sanitize** (*bool, optional (defaul False)*) – If set to True molecules will be sanitized. Note that calculating some features (e.g. aromatic interactions) require sanitized molecules.
- ****kwargs** (*dict, optional*) – Keyword arguments can be usaed to specify custom cutoffs and bins (see default values below).
- **bins** (*Default cutoffs and*) –
- **------------------------** –
- **hbond_dist_bins** (*[(2.2, 2.5), (2.5, 3.2), (3.2, 4.0)]*) –
- **hbond_angle_cutoffs** (*[5, 50, 90]*) –
- **splif_contact_bins** (*[(0, 2.0), (2.0, 3.0), (3.0, 4.5)]*) –
- **ecfp_cutoff** (*4.5*) –
- **sybyl_cutoff** (*7.0*) –
- **salt_bridges_cutoff** (*5.0*) –
- **pi_stack_dist_cutoff** (*4.4*) –
- **pi_stack_angle_cutoff** (*30.0*) –
- **cation_pi_dist_cutoff** (*6.5*) –
- **cation_pi_angle_cutoff** (*30.0*) –

featurize(*datapoints: Iterable[Tuple[str, str]] | None = None*, *log_every_n: int = 100*, ***kwargs*) → ndarray

Calculate features for mol/protein complexes. :param datapoints: List of filenames (PDB, SDF, etc.) for ligand molecules and proteins.

Each element should be a tuple of the form (ligand_filename, protein_filename).

Returns
features – Array of features

Return type
np.ndarray

AtomicConvFeaturizer

class AtomicConvFeaturizer(*frag1_num_atoms*, *frag2_num_atoms*, *complex_num_atoms*, *max_num_neighbors*, *neighbor_cutoff*, *strip_hydrogens=True*)

This class computes the featurization that corresponds to AtomicConvModel.

This class computes featurizations needed for AtomicConvModel. Given two molecular structures, it computes a number of useful geometric features. In particular, for each molecule and the global complex, it computes a coordinates matrix of size (N_atoms, 3) where N_atoms is the number of atoms. It also computes a neighbor-list, a dictionary with N_atoms elements where neighbor-list[i] is a list of the atoms the i-th atom has as neighbors. In addition, it computes a z-matrix for the molecule which is an array of shape (N_atoms,) that contains the atomic number of that atom.

Since the featurization computes these three quantities for each of the two molecules and the complex, a total of 9 quantities are returned for each complex. Note that for efficiency, fragments of the molecules can be provided rather than the full molecules themselves.

__init__(*frag1_num_atoms*, *frag2_num_atoms*, *complex_num_atoms*, *max_num_neighbors*, *neighbor_cutoff*, *strip_hydrogens=True*)

Parameters

- **frag1_num_atoms** (*int*) – Maximum number of atoms in fragment 1.
- **frag2_num_atoms** (*int*) – Maximum number of atoms in fragment 2.
- **complex_num_atoms** (*int*) – Maximum number of atoms in complex of frag1/frag2 together.
- **max_num_neighbors** (*int*) – Maximum number of atoms considered as neighbors.
- **neighbor_cutoff** (*float*) – Maximum distance (angstroms) for two atoms to be considered as neighbors. If more than *max_num_neighbors* atoms fall within this cutoff, the closest *max_num_neighbors* will be used.
- **strip_hydrogens** (*bool (default True)*) – Remove hydrogens before computing featurization.

featurize(*datapoints: Iterable[Tuple[str, str]] | None = None*, *log_every_n: int = 100*, ***kwargs*) → ndarray

Calculate features for mol/protein complexes. :param datapoints: List of filenames (PDB, SDF, etc.) for ligand molecules and proteins.

Each element should be a tuple of the form (ligand_filename, protein_filename).

Returns
features – Array of features

Return type
np.ndarray

3.10.3 Inorganic Crystal Featurizers

These featurizers work with datasets of inorganic crystals.

MaterialCompositionFeaturizer

Material Composition Featurizers are those that work with datasets of crystal compositions with periodic boundary conditions. For inorganic crystal structures, these featurizers operate on chemical compositions (e.g. "MoS2"). They should be applied on systems that have periodic boundary conditions. Composition featurizers are not designed to work with molecules.

ElementPropertyFingerprint

class ElementPropertyFingerprint(*data_source: str = 'matminer'*)

Fingerprint of elemental properties from composition.

Based on the data source chosen, returns properties and statistics (min, max, range, mean, standard deviation, mode) for a compound based on elemental stoichiometry. E.g., the average electronegativity of atoms in a crystal structure. The chemical fingerprint is a vector of these statistics. For a full list of properties and statistics, see `matminer.featurizers.composition.ElementProperty(data_source).feature_labels()`.

This featurizer requires the optional dependencies pymatgen and matminer. It may be useful when only crystal compositions are available (and not 3D coordinates).

See references **[1]_**, **[2]_**,[3],[4] for more details.

References

Examples

```
>>> import deepchem as dc
>>> import pymatgen as mg
>>> comp = mg.core.Composition("Fe2O3")
>>> featurizer = dc.feat.ElementPropertyFingerprint()
>>> features = featurizer.featurize([comp])
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(65,)
```

Note: This class requires matminer and Pymatgen to be installed. *NaN* feature values are automatically converted to 0 by this featurizer.

__init__(*data_source: str = 'matminer'*)

Parameters

data_source (*str of "matminer", "magpie" or "deml" (default "matminer")*) – Source for element property data.

[3] Matminer: Ward, L. et al. Comput. Mater. Sci. 152, 60-69 (2018).
[4] Pymatgen: Ong, S.P. et al. Comput. Mater. Sci. 68, 314-319 (2013).

featurize(*datapoints: Iterable[str] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal compositions.

Parameters

- **datapoints** (*Iterable[str]*) – Iterable sequence of composition strings, e.g. "MoS2".
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *compositions*.

Return type

np.ndarray

ElemNetFeaturizer

class ElemNetFeaturizer

Fixed size vector of length 86 containing raw fractional elemental compositions in the compound. The 86 chosen elements are based on the original implementation at https://github.com/NU-CUCIS/ElemNet.

Returns a vector containing fractional compositions of each element in the compound.

References

Examples

```
>>> import deepchem as dc
>>> comp = "Fe2O3"
>>> featurizer = dc.feat.ElemNetFeaturizer()
>>> features = featurizer.featurize([comp])
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape
(86,)
>>> round(sum(features[0]))
1
```

Note: This class requires Pymatgen to be installed.

get_vector(*comp: DefaultDict*) → ndarray | None

Converts a dictionary containing element names and corresponding compositional fractions into a vector of fractions.

Parameters

comp (*collections.defaultdict object*) – Dictionary mapping element names to fractional compositions.

Returns

fractions – Vector of fractional compositions of each element.

Return type

np.ndarray

MaterialStructureFeaturizer

Material Structure Featurizers are those that work with datasets of crystals with periodic boundary conditions. For inorganic crystal structures, these featurizers operate on pymatgen.Structure objects, which include a lattice and 3D coordinates that specify a periodic crystal structure. They should be applied on systems that have periodic boundary conditions. Structure featurizers are not designed to work with molecules.

SineCoulombMatrix

class SineCoulombMatrix(*max_atoms: int = 100*, *flatten: bool = True*)

Calculate sine Coulomb matrix for crystals.

A variant of Coulomb matrix for periodic crystals.

The sine Coulomb matrix is identical to the Coulomb matrix, except that the inverse distance function is replaced by the inverse of sin**2 of the vector between sites which are periodic in the dimensions of the crystal lattice.

Features are flattened into a vector of matrix eigenvalues by default for ML-readiness. To ensure that all feature vectors are equal length, the maximum number of atoms (eigenvalues) in the input dataset must be specified.

This featurizer requires the optional dependencies pymatgen and matminer. It may be useful when crystal structures with 3D coordinates are available.

See [1]_ for more details.

References

Examples

```
>>> import deepchem as dc
>>> import pymatgen as mg
>>> lattice = mg.core.Lattice.cubic(4.2)
>>> structure = mg.core.Structure(lattice, ["Cs", "Cl"], [[0, 0, 0], [0.5, 0.5, 0.
↪5]])
>>> featurizer = dc.feat.SineCoulombMatrix(max_atoms=2)
>>> features = featurizer.featurize([structure])
>>> type(features[0])
<class 'numpy.ndarray'>
>>> features[0].shape # (max_atoms,)
(2,)
```

Note: This class requires matminer and Pymatgen to be installed.

__init__(*max_atoms: int = 100*, *flatten: bool = True*)

Parameters

- **max_atoms** (*int (default 100)*) – Maximum number of atoms for any crystal in the dataset. Used to pad the Coulomb matrix.
- **flatten** (*bool (default True)*) – Return flattened vector of matrix eigenvalues.

featurize(*datapoints: Iterable[Dict[str, Any] | Any] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal structures.

Parameters

- **datapoints** (*Iterable[Union[Dict, pymatgen.core.Structure]]*) – Iterable sequence of pymatgen structure dictionaries or pymatgen.core.Structure. Please confirm the dictionary representations of pymatgen.core.Structure from https://pymatgen.org/pymatgen.core.structure.html.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

CGCNNFeaturizer

class CGCNNFeaturizer(*radius: float = 8.0*, *max_neighbors: float = 12*, *step: float = 0.2*)

Calculate structure graph features for crystals.

Based on the implementation in Crystal Graph Convolutional Neural Networks (CGCNN). The method constructs a crystal graph representation including atom features and bond features (neighbor distances). Neighbors are determined by searching in a sphere around atoms in the unit cell. A Gaussian filter is applied to neighbor distances. All units are in angstrom.

This featurizer requires the optional dependency pymatgen. It may be useful when 3D coordinates are available and when using graph network models and crystal graph convolutional networks.

See [1]_ for more details.

References

Examples

```
>>> import deepchem as dc
>>> import pymatgen as mg
>>> featurizer = dc.feat.CGCNNFeaturizer()
>>> lattice = mg.core.Lattice.cubic(4.2)
>>> structure = mg.core.Structure(lattice, ["Cs", "Cl"], [[0, 0, 0], [0.5, 0.5, 0.
↪5]])
>>> features = featurizer.featurize([structure])
>>> feature = features[0]
>>> print(type(feature))
<class 'deepchem.feat.graph_data.GraphData'>
```

Note: This class requires Pymatgen to be installed.

`__init__`(*radius: float = 8.0*, *max_neighbors: float = 12*, *step: float = 0.2*)

Parameters

- **radius** (*float (default 8.0)*) – Radius of sphere for finding neighbors of atoms in unit cell.
- **max_neighbors** (*int (default 12)*) – Maximum number of neighbors to consider when constructing graph.
- **step** (*float (default 0.2)*) – Step size for Gaussian filter. This value is used when building edge features.

`featurize`(*datapoints: Iterable[Dict[str, Any] | Any] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal structures.

Parameters

- **datapoints** (*Iterable[Union[Dict, pymatgen.core.Structure]]*) – Iterable sequence of pymatgen structure dictionaries or pymatgen.core.Structure. Please confirm the dictionary representations of pymatgen.core.Structure from https://pymatgen.org/pymatgen.core.structure.html.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

LCNNFeaturizer

class LCNNFeaturizer(*structure: Any*, *aos: List[str]*, *pbc: List[bool]*, *ns: int = 1*, *na: int = 1*, *cutoff: float = 6.0*)

Calculates the 2-D Surface graph features in 6 different permutations-

Based on the implementation of Lattice Graph Convolution Neural Network (LCNN). This method produces the Atom wise features (One Hot Encoding) and Adjacent neighbour in the specified order of permutations. Neighbors are determined by first extracting a site local environment from the primitive cell, and perform graph matching and distance matching to find neighbors. First, the template of the Primitive cell needs to be defined along with periodic boundary conditions and active and spectator site details. structure(Data Point i.e different configuration of adsorbate atoms) is passed for featurization.

This particular featurization produces a regular-graph (equal number of Neighbors) along with its permutation in 6 symmetric axis. This transformation can be applied when orderering of neighboring of nodes around a site play an important role in the propert predictions. Due to consideration of local neighbor environment, this current implementation would be fruitful in finding neighbors for calculating formation energy of adbsorption tasks where the local. Adsorption turns out to be important in many applications such as catalyst and semiconductor design.

The permuted neighbors are calculated using the Primitive cells i.e periodic cells in all the data points are built via lattice transformation of the primitive cell.

Primitive cell Format:

1. **Pymatgen structure object with site_properties key value**
   - **“SiteTypes” mentioning if it is a active site “A1” or spectator**
     site “S1”.

2. ns , the number of spectator types elements. For “S1” its 1.
3. na , the number of active types elements. For “A1” its 1.
4. aos, the different species of active elements “A1”.
5. pbc, the periodic boundary conditions.

Data point Structure Format(Configuration of Atoms):

1. **Pymatgen structure object with site_properties with following key value.**
    - **“SiteTypes”, mentioning if it is a active site “A1” or spectator** site “S1”.
    - “oss”, different occupational sites. For spectator sites make it -1.

It is highly recommended that cells of data are directly redefined from the primitive cell, specifically, the relative coordinates between sites are consistent so that the lattice is non-deviated.

References

Examples

```
>>> import deepchem as dc
>>> from pymatgen.core import Structure
>>> import numpy as np
>>> PRIMITIVE_CELL = {
...   "lattice": [[2.818528, 0.0, 0.0],
...               [-1.409264, 2.440917, 0.0],
...               [0.0, 0.0, 25.508255]],
...   "coords": [[0.66667, 0.33333, 0.090221],
...              [0.33333, 0.66667, 0.18043936],
...              [0.0, 0.0, 0.27065772],
...              [0.66667, 0.33333, 0.36087608],
...              [0.33333, 0.66667, 0.45109444],
...              [0.0, 0.0, 0.49656991]],
...   "species": ['H', 'H', 'H', 'H', 'H', 'He'],
...   "site_properties": {'SiteTypes': ['S1', 'S1', 'S1', 'S1', 'S1', 'A1']}
... }
>>> PRIMITIVE_CELL_INF0 = {
...    "cutoff": np.around(6.00),
...    "structure": Structure(**PRIMITIVE_CELL),
...    "aos": ['1', '0', '2'],
...    "pbc": [True, True, False],
...    "ns": 1,
...    "na": 1
... }
>>> DATA_POINT = {
...   "lattice": [[1.409264, -2.440917, 0.0],
...               [4.227792, 2.440917, 0.0],
...               [0.0, 0.0, 23.17559]],
...   "coords": [[0.0, 0.0, 0.099299],
...              [0.0, 0.33333, 0.198598],
...              [0.5, 0.16667, 0.297897],
...              [0.0, 0.0, 0.397196],
```

(continues on next page)

(continued from previous page)

```
...                 [0.0, 0.33333, 0.496495],
...                 [0.5, 0.5, 0.099299],
...                 [0.5, 0.83333, 0.198598],
...                 [0.0, 0.66667, 0.297897],
...                 [0.5, 0.5, 0.397196],
...                 [0.5, 0.83333, 0.496495],
...                 [0.0, 0.66667, 0.54654766],
...                 [0.5, 0.16667, 0.54654766]],
...   "species": ['H', 'H', 'H', 'H', 'H', 'H',
...                'H', 'H', 'H', 'H', 'He', 'He'],
...   "site_properties": {
...     "SiteTypes": ['S1', 'S1', 'S1', 'S1', 'S1',
...                   'S1', 'S1', 'S1', 'S1', 'S1',
...                   'A1', 'A1'],
...     "oss": ['-1', '-1', '-1', '-1', '-1', '-1',
...             '-1', '-1', '-1', '-1', '0', '2']
...                   }
... }
>>> featuriser = dc.feat.LCNNFeaturizer(**PRIMITIVE_CELL_INF0)
>>> print(type(featuriser._featurize(Structure(**DATA_POINT))))
<class 'deepchem.feat.graph_data.GraphData'>
```

Notes

This Class requires pymatgen , networkx , scipy installed.

`__init__`(*structure: Any*, *aos: List[str]*, *pbc: List[bool]*, *ns: int = 1*, *na: int = 1*, *cutoff: float = 6.0*)

Parameters

- **structure** (*: PymatgenStructure*) – Pymatgen Structure object of the primitive cell used for calculating neighbors from lattice transformations.It also requires site_properties attribute with "Sitetypes"(Active or spectator site).
- **aos** (*List[str]*) – A list of all the active site species. For the Pt, N, NO configuration set it as ['0', '1', '2']
- **pbc** (*List[bool]*) – Periodic Boundary Condition
- **ns** (*int (default 1)*) – The number of spectator types elements. For "S1" its 1.
- **na** (*int (default 1)*) – the number of active types elements. For "A1" its 1.
- **cutoff** (*float (default 6.00)*) – Cutoff of radius for getting local environment.Only used down to 2 digits.

`featurize`(*datapoints: Iterable[Dict[str, Any] | Any] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal structures.

Parameters

- **datapoints** (*Iterable[Union[Dict, pymatgen.core.Structure]]*) – Iterable sequence of pymatgen structure dictionaries or pymatgen.core.Structure. Please confirm the dictionary representations of pymatgen.core.Structure from https://pymatgen.org/pymatgen.core.structure.html.

- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

3.10.4 Biological Sequence Featurizers

These featurizers work with biological sequences.

SAMFeaturizer

class SAMFeaturizer(*max_records=None*)

Featurizes SAM files, that store biological sequences aligned to a reference sequence. This class extracts Query Name, Query Sequence, Query Length, Reference Name,Reference Start, CIGAR and Mapping Quality of each read in a SAM file.

This is the default featurizer used by SAMLoader, and it extracts the following fields from each read in each SAM file in the given order:- - Column 0: Query Name - Column 1: Query Sequence - Column 2: Query Length - Column 3: Reference Name - Column 4: Reference Start - Column 5: CIGAR - Column 6: Mapping Quality

Examples

```
>>> from deepchem.data.data_loader import SAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.sam'
>>> featurizer = dc.feat.SAMFeaturizer()
>>> features = featurizer.featurize(inputs)
>>> type(features[0])
<class 'numpy.ndarray'>
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*max_records=None*)

Initialize SAMFeaturizer.

Parameters
max_records (*int or None, optional*) – The maximum number of records to extract from the SAM file. If None, all records will be extracted.

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.

- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns
A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

BAMFeaturizer

class BAMFeaturizer(*max_records=None*)

Featurizes BAM files, that are compressed binary representations of SAM (Sequence Alignment Map) files. This class extracts Query Name, Query Sequence, Query Length, Reference Name, Reference Start, CIGAR and Mapping Quality of the alignment in the BAM file.

This is the default featurizer used by BAMLoader, and it extracts the following fields from each read in each BAM file in the given order:- - Column 0: Query Name - Column 1: Query Sequence - Column 2: Query Length - Column 3: Reference Name - Column 4: Reference Start - Column 5: CIGAR - Column 6: Mapping Quality

Examples

```
>>> from deepchem.data.data_loader import BAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.bam'
>>> featurizer = dc.feat.BAMFeaturizer()
>>> features = featurizer.featurize(inputs)
>>> type(features[0])
<class 'numpy.ndarray'>
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*max_records=None*)

Initialize BAMFeaturizer.

Parameters
max_records (*int or None, optional*) – The maximum number of records to extract from the BAM file. If None, all records will be extracted.

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns
A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

CRAMFeaturizer

class CRAMFeaturizer(*max_records=None*)

Featurizes CRAM files, that are compressed columnar file format for storing biological sequences aligned to a reference sequence. This class extracts Query Name, Query Sequence, Query Length, Reference Name, Reference Start, CIGAR and Mapping Quality of the alignment in the CRAM file.

This is the default featurizer used by CRAMLoader, and it extracts the following fields from each read in each CRAM file in the given order:- - Column 0: Query Name - Column 1: Query Sequence - Column 2: Query Length - Column 3: Reference Name - Column 4: Reference Start - Column 5: CIGAR - Column 6: Mapping Quality

Examples

```
>>> from deepchem.data.data_loader import CRAMLoader
>>> import deepchem as dc
>>> inputs = 'deepchem/data/tests/example.cram'
>>> featurizer = dc.feat.CRAMFeaturizer()
>>> features = featurizer.featurize(inputs)
>>> type(features[0])
<class 'numpy.ndarray'>
```

Note: This class requires pysam to be installed. Pysam can be used with Linux or MacOS X. To use Pysam on Windows, use Windows Subsystem for Linux(WSL).

__init__(*max_records=None*)

Initialize CRAMFeaturizer.

Parameters
max_records (*int or None, optional*) – The maximum number of records to extract from the CRAM file. If None, all records will be extracted.

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns
A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

3.10.5 Molecule Tokenizers

A tokenizer is in charge of preparing the inputs for a natural language processing model. For many scientific applications, it is possible to treat inputs as "words"/"sentences" and use NLP methods to make meaningful predictions. For example, SMILES strings or DNA sequences have grammatical structure and can be usefully modeled with NLP techniques. DeepChem provides some scientifically relevant tokenizers for use in different applications. These tokenizers are based on those from the Huggingface transformers library (which DeepChem tokenizers inherit from).

The base classes PreTrainedTokenizer and PreTrainedTokenizerFast implements the common methods for encoding string inputs in model inputs and instantiating/saving python tokenizers either from a local file or directory or from a pretrained tokenizer provided by the library (downloaded from HuggingFace's AWS S3 repository).

PreTrainedTokenizer (transformers.PreTrainedTokenizer) thus implements the main methods for using all the tokenizers:

- Tokenizing (spliting strings in sub-word token strings), converting tokens strings to ids and back, and encoding/decoding (i.e. tokenizing + convert to integers)
- Adding new tokens to the vocabulary in a way that is independent of the underlying structure (BPE, SentencePiece…)
- Managing special tokens like mask, beginning-of-sentence, etc tokens (adding them, assigning them to attributes in the tokenizer for easy access and making sure they are not split during tokenization)

BatchEncoding holds the output of the tokenizer's encoding methods (__call__, encode_plus and batch_encode_plus) and is derived from a Python dictionary. When the tokenizer is a pure python tokenizer, this class behave just like a standard python dictionary and hold the various model inputs computed by these methodes (input_ids, attention_mask…). For more details on the base tokenizers which the DeepChem tokenizers inherit from, please refer to the following: HuggingFace tokenizers docs

Tokenization methods on string-based corpuses in the life sciences are becoming increasingly popular for NLP-based applications to chemistry and biology. One such example is ChemBERTa, a transformer for molecular property prediction. DeepChem offers a tutorial for utilizing ChemBERTa using an alternate tokenizer, a Byte-Piece Encoder, which can be found here.

SmilesTokenizer

The `dc.feat.SmilesTokenizer` module inherits from the BertTokenizer class in transformers. It runs a WordPiece tokenization algorithm over SMILES strings using the tokenisation SMILES regex developed by Schwaller et. al.

The SmilesTokenizer employs an atom-wise tokenization strategy using the following Regex expression:

```
SMI_REGEX_PATTERN = "(\[[^\]]+]|Br?|Cl?|N|O|S|P|F|I|b|c|n|o|s|p|\(|\)|\.|=|#||\+|\\\\\/
↪|:||@|\?|>|\*|\$|\%[0-9]{2}|[0-9])"
```

To use, please install the transformers package using the following pip command:

```
pip install transformers
```

References:

- RXN Mapper: Unsupervised Attention-Guided Atom-Mapping
- Molecular Transformer: Unsupervised Attention-Guided Atom-Mapping

class SmilesTokenizer(*vocab_file: str = ''*, ***kwargs*)

Creates the SmilesTokenizer class. The tokenizer heavily inherits from the BertTokenizer implementation found in Huggingface's transformers library. It runs a WordPiece tokenization algorithm over SMILES strings using the tokenisation SMILES regex developed by Schwaller et. al.

Please see https://github.com/huggingface/transformers and https://github.com/rxn4chemistry/rxnfp for more details.

Examples

```
>>> from deepchem.feat.smiles_tokenizer import SmilesTokenizer
>>> current_dir = os.path.dirname(os.path.realpath(__file__))
>>> vocab_path = os.path.join(current_dir, 'tests/data', 'vocab.txt')
>>> tokenizer = SmilesTokenizer(vocab_path)
>>> print(tokenizer.encode("CC(=O)OC1=CC=CC=C1C(=O)O"))
[12, 16, 16, 17, 22, 19, 18, 19, 16, 20, 22, 16, 16, 22, 16, 16, 22, 16, 20, 16, 17,
↪ 22, 19, 18, 19, 13]
```

References

Note: This class requires huggingface's transformers and tokenizers libraries to be installed.

`__init__`(*vocab_file: str = ''*, ***kwargs*)

Constructs a SmilesTokenizer.

Parameters

vocab_file (*str*) – Path to a SMILES character per line vocabulary file. Default vocab file is found in deepchem/feat/tests/data/vocab.txt

property vocab_size

Size of the base vocabulary (without the added tokens).

Type

int

`convert_tokens_to_string`(*tokens: List[str]*)

Converts a sequence of tokens (string) in a single string.

Parameters

tokens (*List[str]*) – List of tokens for a given string sequence.

Returns

out_string – Single string from combined tokens.

Return type

str

`add_special_tokens_ids_single_sequence`(*token_ids: List[int | None]*)

Adds special tokens to the a sequence for sequence classification tasks.

A BERT sequence has the following format: [CLS] X [SEP]

Parameters

token_ids (*list[int]*) – list of tokenized input ids. Can be obtained using the encode or encode_plus methods.

`add_special_tokens_single_sequence`(*tokens: List[str]*)

Adds special tokens to the a sequence for sequence classification tasks. A BERT sequence has the following format: [CLS] X [SEP]

Parameters
`tokens` (*List[str]*) – List of tokens for a given string sequence.

`add_special_tokens_ids_sequence_pair`(*token_ids_0: List[int | None]*, *token_ids_1: List[int | None]*) → List[int | None]

Adds special tokens to a sequence pair for sequence classification tasks. A BERT sequence pair has the following format: [CLS] A [SEP] B [SEP]

Parameters

- `token_ids_0` (*List[int]*) – List of ids for the first string sequence in the sequence pair (A).
- `token_ids_1` (*List[int]*) – List of tokens for the second string sequence in the sequence pair (B).

`add_padding_tokens`(*token_ids: List[int | None]*, *length: int*, *right: bool = True*) → List[int | None]

Adds padding tokens to return a sequence of length max_length. By default padding tokens are added to the right of the sequence.

Parameters

- `token_ids` (*list[optional[int]]*) – list of tokenized input ids. Can be obtained using the encode or encode_plus methods.
- `length` (*int*) – TODO
- `right` (*bool, default True*) – TODO

Returns
TODO

Return type
List[int]

`save_vocabulary`(*save_directory: str*, *filename_prefix: str | None = None*)

Save the tokenizer vocabulary to a file.

Parameters
`vocab_path` (*obj: str*) – The directory in which to save the SMILES character per line vocabulary file. Default vocab file is found in deepchem/feat/tests/data/vocab.txt

Returns
vocab_file – Paths to the files saved. typle with string to a SMILES character per line vocabulary file. Default vocab file is found in deepchem/feat/tests/data/vocab.txt

Return type
Tuple

BasicSmilesTokenizer

The `dc.feat.BasicSmilesTokenizer` module uses a regex tokenization pattern to tokenise SMILES strings. The regex is developed by Schwaller et. al. The tokenizer is to be used on SMILES in cases where the user wishes to not rely on the transformers API.

References:

- Molecular Transformer: Unsupervised Attention-Guided Atom-Mapping

class BasicSmilesTokenizer(*regex_pattern: str =* `'(\\[[^\\]]+]|Br?|Cl?|N|O|S|P|F|I|b|c|n|o|s|p|\\(|\\)|\\.|=|#|-|\\+|\\\\|\\/|:|~|@|\\?|>>?|\\*|\\$|\\%[0-9]{2}|[0-9])'`)

Run basic SMILES tokenization using a regex pattern developed by Schwaller et. al. This tokenizer is to be used when a tokenizer that does not require the transformers library by HuggingFace is required.

Examples

```
>>> from deepchem.feat.smiles_tokenizer import BasicSmilesTokenizer
>>> tokenizer = BasicSmilesTokenizer()
>>> print(tokenizer.tokenize("CC(=O)OC1=CC=CC=C1C(=O)O"))
['C', 'C', '(', '=', 'O', ')', 'O', 'C', '1', '=', 'C', 'C', '=', 'C', 'C', '=', 'C
↪', '1', 'C', '(', '=', 'O', ')', 'O']
```

References

__init__(*regex_pattern: str =* `'(\\[[^\\]]+]|Br?|Cl?|N|O|S|P|F|I|b|c|n|o|s|p|\\(|\\)|\\.|=|#|-|\\+|\\\\|\\/|:|~|@|\\?|>>?|\\*|\\$|\\%[0-9]{2}|[0-9])'`)

Constructs a BasicSMILESTokenizer.

Parameters

regex (*string*) – SMILES token regex

tokenize(*text*)

Basic Tokenization of a SMILES.

HuggingFaceFeaturizer

class HuggingFaceFeaturizer(*tokenizer: transformers.tokenization_utils_fast.PreTrainedTokenizerFast*)

Wrapper class that wraps HuggingFace tokenizers as DeepChem featurizers

The *HuggingFaceFeaturizer* wrapper provides a wrapper around Hugging Face tokenizers allowing them to be used as DeepChem featurizers. This might be useful in scenarios where user needs to use a hugging face tokenizer when loading a dataset.

Example

```
>>> from deepchem.feat import HuggingFaceFeaturizer
>>> from transformers import RobertaTokenizerFast
>>> hf_tokenizer = RobertaTokenizerFast.from_pretrained("seyonec/PubChem10M_SMILES_
↪BPE_60k")
>>> featurizer = HuggingFaceFeaturizer(tokenizer=hf_tokenizer)
>>> result = featurizer.featurize(['CC(=O)C'])
```

__init__(*tokenizer: transformers.tokenization_utils_fast.PreTrainedTokenizerFast*)

Initializes a tokenizer wrapper

Parameters

tokenizer (*transformers.tokenization_utils_fast.PreTrainedTokenizerFast*) – The tokenizer to use for featurization

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (`Iterable[Any]`) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **log_every_n** (`int, default 1000`) – Logs featurization progress every *log_every_n* steps.

Returns

A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

GroverAtomVocabTokenizer

class GroverAtomVocabTokenizer(*fname: str*)

Grover Atom Vocabulary Tokenizer

The Grover Atom vocab tokenizer is used for tokenizing an atom using a vocabulary generated by GroverAtomVocabularyBuilder.

Example

```
>>> import tempfile
>>> import deepchem as dc
>>> from deepchem.feat.vocabulary_builders.grover_vocab import␣
↪GroverAtomVocabularyBuilder
>>> file = tempfile.NamedTemporaryFile()
>>> dataset = dc.data.NumpyDataset(X=[['CC(=O)C'], ['CCC']])
>>> vocab = GroverAtomVocabularyBuilder()
>>> vocab.build(dataset)
>>> vocab.save(file.name)  # build and save the vocabulary
>>> atom_tokenizer = GroverAtomVocabTokenizer(file.name)
>>> mol = Chem.MolFromSmiles('CC(=O)C')
>>> atom_tokenizer.featurize([(mol, mol.GetAtomWithIdx(0))])[0]
2
```

Parameters

fname (`str`) – Filename of vocabulary generated by GroverAtomVocabularyBuilder

__init__(*fname: str*)

GroverBondVocabTokenizer

class GroverBondVocabTokenizer(*fname: str*)

Grover Bond Vocabulary Tokenizer

The Grover Bond vocab tokenizer is used for tokenizing a bond using a vocabulary generated by GroverBondVocabularyBuilder.

Example

```
>>> import tempfile
>>> import deepchem as dc
>>> from deepchem.feat.vocabulary_builders.grover_vocab import␣
↪GroverBondVocabularyBuilder
>>> file = tempfile.NamedTemporaryFile()
>>> dataset = dc.data.NumpyDataset(X=[['CC(=O)C'], ['CCC']])
>>> vocab = GroverBondVocabularyBuilder()
>>> vocab.build(dataset)
>>> vocab.save(file.name)  # build and save the vocabulary
>>> bond_tokenizer = GroverBondVocabTokenizer(file.name)
>>> mol = Chem.MolFromSmiles('CC(=O)C')
>>> bond_tokenizer.featurize([(mol, mol.GetBondWithIdx(0))])[0]
2
```

Parameters

fname (*str*) – Filename of vocabulary generated by GroverAtomVocabularyBuilder

__init__(*fname: str*)

3.10.6 Vocabulary Builders

Tokenizers uses a vocabulary to tokenize the datapoint. To build a vocabulary, an algorithm which generates vocabulary from a corpus is required. A corpus is usually a collection of molecules, DNA sequences etc. DeepChem provides the following algorithms to build vocabulary from a corpus. A vocabulary builder is not a featurizer. It is an utility which helps the tokenizers to featurize datapoints.

class GroverAtomVocabularyBuilder(*max_size: int | None = None*)

Atom Vocabulary Builder for Grover

This module can be used to generate atom vocabulary from SMILES strings for the GROVER pretraining task. For each atom in a molecule, the vocabulary context is the node-edge-count of the atom where node is the neighboring atom, edge is the type of bond (single bond or double bound) and count is the number of such node-edge pairs for the atom in its neighborhood. For example, for the molecule 'CC(=O)C', the context of the first carbon atom is C-SINGLE1 because it's neighbor is C atom, the type of bond is SINGLE bond and the count of such bonds is 1. The context of the second carbon atom is C-SINGLE2 and O-DOUBLE1 because it is connected to two carbon atoms by a single bond and 1 O atom by a double bond. The vocabulary of an atom is then computed as the *atom-symbol_contexts* where the contexts are sorted in alphabetical order when there are multiple contexts. For example, the vocabulary of second C is *C_C-SINGLE2_O-DOUBLE1*. The algorithm enumerates vocabulary of all atoms in the dataset and makes a vocabulary to index mapping by sorting the vocabulary by frequency and then alphabetically.

The algorithm enumerates vocabulary of all atoms in the dataset and makes a vocabulary to index mapping by sorting the vocabulary by frequency and then alphabetically. The *max_size* parameter can be used for setting the size of the vocabulary. When this parameter is set, the algorithm stops adding new words to the index when the vocabulary size reaches *max_size*.

Parameters
max_size (*int (optional)*) – Maximum size of vocabulary

Example

```
>>> import tempfile
>>> import deepchem as dc
>>> from rdkit import Chem
>>> file = tempfile.NamedTemporaryFile()
>>> dataset = dc.data.NumpyDataset(X=[['CCC'], ['CC(=O)C']])
>>> vocab = GroverAtomVocabularyBuilder()
>>> vocab.build(dataset)
>>> vocab.stoi
{'<pad>': 0, '<other>': 1, 'C_C-SINGLE1': 2, 'C_C-SINGLE2': 3, 'C_C-SINGLE2_O-
↪DOUBLE1': 4, 'O_C-DOUBLE1': 5}
>>> vocab.save(file.name)
>>> loaded_vocab = GroverAtomVocabularyBuilder.load(file.name)
>>> mol = Chem.MolFromSmiles('CC(=O)C')
>>> loaded_vocab.encode(mol, mol.GetAtomWithIdx(1))
4
```

Reference

__init__(*max_size: int | None = None*)

build(*dataset: Dataset*, *log_every_n: int = 1000*) → None
Builds vocabulary

Parameters

- **dataset** (*dc.data.Dataset*) – A dataset object with SMILEs strings in X attribute.
- **log_every_n** (*int, default 1000*) – Logs vocabulary building progress every *log_every_n* steps.

build_from_csv(*csv_path: str*, *smiles_field: str*, *log_every_n: int = 1000*) → None
Builds vocabulary from csv file

Parameters

- **csv_path** (*str*) – Path to csv file containing smiles string
- **smiles_field** (*str*) – Name of column containing smiles string
- **log_every_n** (*int, default 1000*) – Logs vocabulary building progress every *log_every_n* steps.

save(*fname: str*) → None
Saves a vocabulary in json format

Parameter

fname: str
Filename to save vocabulary

classmethod load(*fname: str*) → *GroverAtomVocabularyBuilder*

Loads vocabulary from the specified json file

Parameters
fname (*str*) – JSON file containing vocabulary

Returns
vocab – A grover atom vocabulary builder which can be used for encoding

Return type
GroverAtomVocabularyBuilder

static atom_to_vocab(*mol: Any*, *atom: Any*) → str

Convert atom to vocabulary.

Parameters

- **mol** (*RDKitMol*) – an molecule object
- **atom** (*RDKitAtom*) – the target atom.

Returns
vocab – The generated atom vocabulary with its contexts.

Return type
str

Example

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('[C@@H](C)C(=O)O')
>>> GroverAtomVocabularyBuilder.atom_to_vocab(mol, mol.GetAtomWithIdx(0))
'C_C-SINGLE2'
>>> GroverAtomVocabularyBuilder.atom_to_vocab(mol, mol.GetAtomWithIdx(3))
'O_C-DOUBLE1'
```

encode(*mol: Any*, *atom: Any*) → str

Encodes an atom in a molecule

Parameter

mol: RDKitMol
An RDKitMol object

atom: RDKitAtom
An atom in the molecule

returns
vocab – The vocabulary of the atom in the molecule.

rtype
str

class GroverAtomVocabularyBuilder(*max_size: int | None = None*)

Atom Vocabulary Builder for Grover

This module can be used to generate atom vocabulary from SMILES strings for the GROVER pretraining task. For each atom in a molecule, the vocabulary context is the node-edge-count of the atom where node is the neighboring atom, edge is the type of bond (single bond or double bound) and count is the number of such node-edge pairs for the atom in its neighborhood. For example, for the molecule 'CC(=O)C', the context of the first carbon atom is C-SINGLE1 because it's neighbor is C atom, the type of bond is SINGLE bond and the count of such bonds is 1. The context of the second carbon atom is C-SINGLE2 and O-DOUBLE1 because it is connected to two carbon atoms by a single bond and 1 O atom by a double bond. The vocabulary of an atom is then computed as the *atom-symbol_contexts* where the contexts are sorted in alphabetical order when there are multiple contexts. For example, the vocabulary of second C is *C_C-SINGLE2_O-DOUBLE1*. The algorithm enumerates vocabulary of all atoms in the dataset and makes a vocabulary to index mapping by sorting the vocabulary by frequency and then alphabetically.

The algorithm enumerates vocabulary of all atoms in the dataset and makes a vocabulary to index mapping by sorting the vocabulary by frequency and then alphabetically. The *max_size* parameter can be used for setting the size of the vocabulary. When this parameter is set, the algorithm stops adding new words to the index when the vocabulary size reaches *max_size*.

Parameters
max_size (*int (optional)*) – Maximum size of vocabulary

Example

```
>>> import tempfile
>>> import deepchem as dc
>>> from rdkit import Chem
>>> file = tempfile.NamedTemporaryFile()
>>> dataset = dc.data.NumpyDataset(X=[['CCC'], ['CC(=O)C']])
>>> vocab = GroverAtomVocabularyBuilder()
>>> vocab.build(dataset)
>>> vocab.stoi
{'<pad>': 0, '<other>': 1, 'C_C-SINGLE1': 2, 'C_C-SINGLE2': 3, 'C_C-SINGLE2_O-
↪DOUBLE1': 4, 'O_C-DOUBLE1': 5}
>>> vocab.save(file.name)
>>> loaded_vocab = GroverAtomVocabularyBuilder.load(file.name)
>>> mol = Chem.MolFromSmiles('CC(=O)C')
>>> loaded_vocab.encode(mol, mol.GetAtomWithIdx(1))
4
```

Reference

__init__(*max_size: int | None = None*)

build(*dataset: Dataset*, *log_every_n: int = 1000*) → None

Builds vocabulary

Parameters

- **dataset** (*dc.data.Dataset*) – A dataset object with SMILEs strings in X attribute.
- **log_every_n** (*int, default 1000*) – Logs vocabulary building progress every *log_every_n* steps.

build_from_csv(*csv_path: str*, *smiles_field: str*, *log_every_n: int = 1000*) → None

Builds vocabulary from csv file

Parameters

- **csv_path** (*str*) – Path to csv file containing smiles string
- **smiles_field** (*str*) – Name of column containing smiles string
- **log_every_n** (*int, default 1000*) – Logs vocabulary building progress every *log_every_n* steps.

save(*fname: str*) → None

Saves a vocabulary in json format

Parameter

fname: str
Filename to save vocabulary

classmethod load(*fname: str*) → *GroverAtomVocabularyBuilder*

Loads vocabulary from the specified json file

Parameters
fname (*str*) – JSON file containing vocabulary

Returns
vocab – A grover atom vocabulary builder which can be used for encoding

Return type
GroverAtomVocabularyBuilder

static atom_to_vocab(*mol: Any*, *atom: Any*) → str

Convert atom to vocabulary.

Parameters

- **mol** (*RDKitMol*) – an molecule object
- **atom** (*RDKitAtom*) – the target atom.

Returns
vocab – The generated atom vocabulary with its contexts.

Return type
str

Example

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('[C@@H](C)C(=O)O')
>>> GroverAtomVocabularyBuilder.atom_to_vocab(mol, mol.GetAtomWithIdx(0))
'C_C-SINGLE2'
>>> GroverAtomVocabularyBuilder.atom_to_vocab(mol, mol.GetAtomWithIdx(3))
'O_C-DOUBLE1'
```

encode(*mol: Any*, *atom: Any*) → str

Encodes an atom in a molecule

Parameter

mol: RDKitMol
An RDKitMol object

atom: RDKitAtom
An atom in the molecule

returns
vocab – The vocabulary of the atom in the molecule.

rtype
str

3.10.7 Sequence Featurizers

PFMFeaturizer

The `dc.feat.PFMFeaturizer` module implements a featurizer for position frequency matrices. This takes in a list of multisequence alignments and returns a list of position frequency matrices.

class PFMFeaturizer(*charset: List[str] = ['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', 'X', 'Z', 'B', 'U', 'O'], max_length: int | None = 100*)

Encodes a list position frequency matrices for a given list of multiple sequence alignments

The default character set is 25 amino acids. If you want to use a different character set, such as nucleotides, simply pass in a list of character strings in the featurizer constructor.

The max_length parameter is the maximum length of the sequences to be featurized. If you want to featurize longer sequences, modify the max_length parameter in the featurizer constructor.

The final row in the position frequency matrix is the unknown set, if there are any characters which are not included in the charset.

Examples

```
>>> from deepchem.feat.sequence_featurizers import PFMFeaturizer
>>> from deepchem.data import NumpyDataset
>>> msa = NumpyDataset(X=[['ABC','BCD'],['AAA','AAB']], ids=[['seq01','seq02'],[
↪'seq11','seq12']])
>>> seqs = msa.X
>>> featurizer = PFMFeaturizer()
>>> pfm = featurizer.featurize(seqs)
>>> pfm.shape
(2, 26, 100)
```

__init__(*charset: List[str] = ['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', 'X', 'Z', 'B', 'U', 'O'], max_length: int | None = 100*)

Initialize featurizer.

Parameters

- **charset** (*List[str] (default CHARSET)*) – A list of strings, where each string is length 1 and unique.

- **max_length** (*int, optional (default 25)*) – Maximum length of sequences to be featurized.

3.10.8 Other Featurizers

BertFeaturizer

class BertFeaturizer(*tokenizer: BertTokenizerFast*)

Bert Featurizer.

Bert Featurizer. The Bert Featurizer is a wrapper class for HuggingFace's BertTokenizerFast. This class intends to allow users to use the BertTokenizer API while remaining inside the DeepChem ecosystem.

Examples

```
>>> from deepchem.feat import BertFeaturizer
>>> from transformers import BertTokenizerFast
>>> tokenizer = BertTokenizerFast.from_pretrained("Rostlab/prot_bert", do_lower_
↪case=False)
>>> featurizer = BertFeaturizer(tokenizer)
>>> feats = featurizer.featurize(['D L I P [MASK] L V T'])
```

Notes

Examples are based on RostLab's ProtBert documentation.

__init__(*tokenizer: BertTokenizerFast*)

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns

A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

RobertaFeaturizer

class RobertaFeaturizer(***kwargs*)

Roberta Featurizer.

The Roberta Featurizer is a wrapper class of the Roberta Tokenizer, which is used by Huggingface's transformers library for tokenizing large corpuses for Roberta Models. Please confirm the details in [1]_.

Please see https://github.com/huggingface/transformers and https://github.com/seyonechithrananda/bert-loves-chemistry for more details.

Examples

```
>>> from deepchem.feat import RobertaFeaturizer
>>> smiles = ["Cn1c(=O)c2c(ncn2C)n(C)c1=O", "CC(=O)N1CN(C(C)=O)C(O)C1O"]
>>> featurizer = RobertaFeaturizer.from_pretrained("seyonec/SMILES_tokenized_
↪PubChem_shard00_160k")
>>> out = featurizer(smiles, add_special_tokens=True, truncation=True)
```

References

Note: This class requires transformers to be installed. RobertaFeaturizer uses dual inheritance with RobertaTokenizerFast in Huggingface for rapid tokenization, as well as DeepChem's MolecularFeaturizer class.

__init__(***kwargs*)

__len__() → int

Size of the full vocabulary with the added tokens.

add_special_tokens(*special_tokens_dict: Dict[str, str | AddedToken]*, *replace_additional_special_tokens=True*) → int

Add a dictionary of special tokens (eos, pad, cls, etc.) to the encoder and link them to class attributes. If special tokens are NOT in the vocabulary, they are added to it (indexed starting from the last index of the current vocabulary).

When adding new tokens to the vocabulary, you should make sure to also resize the token embedding matrix of the model so that its embedding matrix matches the tokenizer.

In order to do that, please use the [~*PreTrainedModel.resize_token_embeddings*] method.

Using *add_special_tokens* will ensure your special tokens can be used in several ways:

- Special tokens can be skipped when decoding using *skip_special_tokens* = *True*.
- Special tokens are carefully handled by the tokenizer (they are never split), similar to *AddedTokens*.
- You can easily refer to special tokens using tokenizer class attributes like *tokenizer.cls_token*. This makes it easy to develop model-agnostic training and fine-tuning scripts.

When possible, special tokens are already registered for provided pretrained models (for instance [*BertTokenizer*] *cls_token* is already registered to be :obj*'[CLS]'* and XLM's one is also registered to be '*</s>*').

Parameters

- **special_tokens_dict** (dictionary *str* to *str* or *tokenizers.AddedToken*) – Keys should be in the list of predefined special attributes: [*bos_token*, *eos_token*, *unk_token*, *sep_token*, *pad_token*, *cls_token*, *mask_token*, *additional_special_tokens*].

  Tokens are only added if they are not already in the vocabulary (tested by checking if the tokenizer assign the index of the *unk_token* to them).

- **replace_additional_special_tokens** (*bool*, *optional*,, defaults to *True*) – If *True*, the existing list of additional special tokens will be replaced by the list provided in *special_tokens_dict*. Otherwise, *self._additional_special_tokens* is just extended. In the former case, the tokens will NOT be removed from the tokenizer's full vocabulary - they are only being flagged as non-special tokens. Remember, this only affects which tokens are skipped during decoding, not the *added_tokens_encoder* and *added_tokens_decoder*. This means that the previous *additional_special_tokens* are still added tokens, and will not be split by the model.

Returns
Number of tokens added to the vocabulary.

Return type
int

Examples:

```python # Let's see how to add a new classification token to GPT-2 tokenizer = GPT2Tokenizer.from_pretrained("openai-community/gpt2") model = GPT2Model.from_pretrained("openai-community/gpt2")

special_tokens_dict = {"cls_token": "<CLS>"}

num_added_toks = tokenizer.add_special_tokens(special_tokens_dict) print("We have added", num_added_toks, "tokens") # Notice: resize_token_embeddings expect to receive the full size of the new vocabulary, i.e., the length of the tokenizer. model.resize_token_embeddings(len(tokenizer))

assert tokenizer.cls_token == "<CLS>" ```

**add_tokens**(*new_tokens: str | AddedToken | List[str | AddedToken]*, *special_tokens: bool = False*) → int

Add a list of new tokens to the tokenizer class. If the new tokens are not in the vocabulary, they are added to it with indices starting from length of the current vocabulary and and will be isolated before the tokenization algorithm is applied. Added tokens and tokens from the vocabulary of the tokenization algorithm are therefore not treated in the same way.

Note, when adding new tokens to the vocabulary, you should make sure to also resize the token embedding matrix of the model so that its embedding matrix matches the tokenizer.

In order to do that, please use the [*~PreTrainedModel.resize_token_embeddings*] method.

**Parameters**

- **new_tokens** (*str*, *tokenizers.AddedToken* or a list of *str* or *tokenizers.AddedToken*) – Tokens are only added if they are not already in the vocabulary. *tokenizers.AddedToken* wraps a string token to let you personalize its behavior: whether this token should only match against a single word, whether this token should strip all potential whitespaces on the left side, whether this token should strip all potential whitespaces on the right side, etc.

- **special_tokens** (*bool*, *optional*, defaults to *False*) – Can be used to specify if the token is a special token. This mostly change the normalization behavior (special tokens like CLS or [MASK] are usually not lower-cased for instance).

  See details for *tokenizers.AddedToken* in HuggingFace tokenizers library.
```

Returns
Number of tokens added to the vocabulary.

Return type
int

Examples:

```python # Let's see how to increase the vocabulary of Bert model and tokenizer tokenizer = BertTokenizerFast.from_pretrained("google-bert/bert-base-uncased") model = BertModel.from_pretrained("google-bert/bert-base-uncased")

num_added_toks = tokenizer.add_tokens(["new_tok1", "my_new-tok2"]) print("We have added", num_added_toks, "tokens") # Notice: resize_token_embeddings expect to receive the full size of the new vocabulary, i.e., the length of the tokenizer. model.resize_token_embeddings(len(tokenizer)) ```

`property added_tokens_decoder: Dict[int, AddedToken]`

Returns the added tokens in the vocabulary as a dictionary of index to AddedToken.

**Returns**
The added tokens.

**Return type**
*Dict[str, int]*

`property added_tokens_encoder: Dict[str, int]`

Returns the sorted mapping from string to index. The added tokens encoder is cached for performance optimisation in *self._added_tokens_encoder* for the slow tokenizers.

`property additional_special_tokens: List[str]`

All the additional special tokens you may want to use. Log an error if used while not having been set.

**Type**
*List[str]*

`property additional_special_tokens_ids: List[int]`

Ids of all the additional special tokens in the vocabulary. Log an error if used while not having been set.

**Type**
*List[int]*

`property all_special_ids: List[int]`

List the ids of the special tokens(*'<unk>'*, *'<cls>'*, etc.) mapped to class attributes.

**Type**
*List[int]*

`property all_special_tokens: List[str]`

A list of the unique special tokens (*'<unk>'*, *'<cls>'*, . . . , etc.).

Convert tokens of *tokenizers.AddedToken* type to string.

**Type**
*List[str]*

`property all_special_tokens_extended: List[str | AddedToken]`

All the special tokens (*'<unk>'*, *'<cls>'*, etc.), the order has nothing to do with the index of each tokens. If you want to know the correct indices, check *self.added_tokens_encoder*. We can't create an order anymore as the keys are *AddedTokens* and not *Strings*.

Don't convert tokens of *tokenizers.AddedToken* type to string so they can be used to control more finely how special tokens are tokenized.
```

Type
List[Union[str, tokenizers.AddedToken]]

`apply_chat_template`(*conversation: List[Dict[str, str]] | List[List[Dict[str, str]]] | Conversation*, *chat_template: str | None = None*, *add_generation_prompt: bool = False*, *tokenize: bool = True*, *padding: bool = False*, *truncation: bool = False*, *max_length: int | None = None*, *return_tensors: str | TensorType | None = None*, *return_dict: bool = False*, *tokenizer_kwargs: Dict[str, Any] | None = None*, ***kwargs*) → str | List[int] | List[str] | List[List[int]] | BatchEncoding

Converts a list of dictionaries with *"role"* and *"content"* keys to a list of token ids. This method is intended for use with chat models, and will read the tokenizer's chat_template attribute to determine the format and control tokens to use when converting. When chat_template is None, it will fall back to the default_chat_template specified at the class level.

Parameters

- **conversation** (*Union[List[Dict[str, str]], List[List[Dict[str, str]]], "Conversation"]*) – A list of dicts with "role" and "content" keys, representing the chat history so far.
- **chat_template** (str, *optional*) – A Jinja template to use for this conversion. If this is not passed, the model's default chat template will be used instead.
- **add_generation_prompt** (bool, *optional*) – Whether to end the prompt with the token(s) that indicate the start of an assistant message. This is useful when you want to generate a response from the model. Note that this argument will be passed to the chat template, and so it must be supported in the template for this argument to have any effect.
- **tokenize** (*bool*, defaults to *True*) – Whether to tokenize the output. If *False*, the output will be a string.
- **padding** (*bool*, defaults to *False*) – Whether to pad sequences to the maximum length. Has no effect if tokenize is *False*.
- **truncation** (*bool*, defaults to *False*) – Whether to truncate sequences at the maximum length. Has no effect if tokenize is *False*.
- **max_length** (*int*, *optional*) – Maximum length (in tokens) to use for padding or truncation. Has no effect if tokenize is *False*. If not specified, the tokenizer's *max_length* attribute will be used as a default.
- **return_tensors** (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors of a particular framework. Has no effect if tokenize is *False*. Acceptable values are: - *'tf'*: Return TensorFlow *tf.Tensor* objects. - *'pt'*: Return PyTorch *torch.Tensor* objects. - *'np'*: Return NumPy *np.ndarray* objects. - *'jax'*: Return JAX *jnp.ndarray* objects.
- **return_dict** (*bool*, defaults to *False*) – Whether to return a dictionary with named outputs. Has no effect if tokenize is *False*.
- **(`Dict[str** (*tokenizer_kwargs*) – Any]`, *optional*): Additional kwargs to pass to the tokenizer.
- ****kwargs** – Additional kwargs to pass to the template renderer. Will be accessible by the chat template.

Returns
A list of token ids representing the tokenized chat so far, including control tokens. This output is ready to pass to the model, either directly or via methods like *generate()*. If *return_dict* is set, will return a dict of tokenizer outputs instead.

Return type
Union[List[int], Dict]

`as_target_tokenizer()`

Temporarily sets the tokenizer for encoding the targets. Useful for tokenizer associated to sequence-to-sequence models that need a slightly different processing for the labels.

`property backend_tokenizer: Tokenizer`

The Rust tokenizer used as a backend.

Type
tokenizers.implementations.BaseTokenizer

`batch_decode`(*sequences: List[int] | List[List[int]] | np.ndarray | torch.Tensor | tf.Tensor*, *skip_special_tokens: bool = False*, *clean_up_tokenization_spaces: bool = None*, ***kwargs*) → List[str]

Convert a list of lists of token ids into a list of strings by calling decode.

Parameters

- `sequences` (*Union[List[int], List[List[int]], np.ndarray, torch.Tensor, tf.Tensor]*) – List of tokenized input ids. Can be obtained using the *__call__* method.
- `skip_special_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not to remove special tokens in the decoding.
- `clean_up_tokenization_spaces` (*bool*, *optional*) – Whether or not to clean up the tokenization spaces. If *None*, will default to *self.clean_up_tokenization_spaces*.
- `kwargs` (additional keyword arguments, *optional*) – Will be passed to the underlying model specific decode method.

Returns
The list of decoded sentences.

Return type
List[str]

`batch_encode_plus`(*batch_text_or_text_pairs: List[str] | List[Tuple[str, str]] | List[List[str]] | List[Tuple[List[str], List[str]]] | List[List[int]] | List[Tuple[List[int], List[int]]]*, *add_special_tokens: bool = True*, *padding: bool | str | PaddingStrategy = False*, *truncation: bool | str | TruncationStrategy | None = None*, *max_length: int | None = None*, *stride: int = 0*, *is_split_into_words: bool = False*, *pad_to_multiple_of: int | None = None*, *return_tensors: str | TensorType | None = None*, *return_token_type_ids: bool | None = None*, *return_attention_mask: bool | None = None*, *return_overflowing_tokens: bool = False*, *return_special_tokens_mask: bool = False*, *return_offsets_mapping: bool = False*, *return_length: bool = False*, *verbose: bool = True*, ***kwargs*) → BatchEncoding

Tokenize and prepare for the model a list of sequences or a list of pairs of sequences.

<Tip warning={true}>

This method is deprecated, *__call__* should be used instead.

</Tip>

Parameters

- `batch_text_or_text_pairs` (*List[str]*, *List[Tuple[str, str]]*, *List[List[str]]*, *List[Tuple[List[str], List[str]]]*, and for not-fast tokenizers, also *List[List[int]]*, *List[Tuple[List[int], List[int]]]*) – Batch of sequences or pair of sequences to be encoded. This can be a list of string/string-sequences/int-sequences or a list of pair of string/string-sequences/int-sequence (see details in *encode_plus*).

- `add_special_tokens` (*bool*, *optional*, defaults to *True*) – Whether or not to add special tokens when encoding the sequences. This will use the underlying *PretrainedTokenizerBase.build_inputs_with_special_tokens* function, which defines which tokens are automatically added to the input ids. This is usefull if you want to add *bos* or *eos* tokens automatically.
- `padding` (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *False*) – Activates and controls padding. Accepts the following values:
  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).
  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).
- `truncation` (*bool*, *str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *False*) – Activates and controls truncation. Accepts the following values:
  - *True* or *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *False* or *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- `max_length` (*int*, *optional*) – Controls the maximum length to use by one of the truncation/padding parameters.

  If left unset or set to *None*, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.
- `stride` (*int*, *optional*, defaults to 0) – If set to a number along with *max_length*, the overflowing tokens returned when *return_overflowing_tokens=True* will contain some tokens from the end of the truncated sequence returned to provide some overlap between truncated and overflowing sequences. The value of this argument defines the number of overlapping tokens.
- `is_split_into_words` (*bool*, *optional*, defaults to *False*) – Whether or not the input is already pre-tokenized (e.g., split into words). If set to *True*, the tokenizer assumes the input is already split into words (for instance, by splitting it on whitespace) which it will tokenize. This is useful for NER or token classification.

- `pad_to_multiple_of` (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value. Requires *padding* to be activated. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= *7.5* (Volta).
- `return_tensors` (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:
  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.
- `return_token_type_ids` (*bool*, *optional*) – Whether to return token type IDs. If left to the default, will return the token type IDs according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are token type IDs?](../glossary#token-type-ids)
- `return_attention_mask` (*bool*, *optional*) – Whether to return the attention mask. If left to the default, will return the attention mask according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are attention masks?](../glossary#attention-mask)
- `return_overflowing_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not to return overflowing token sequences. If a pair of sequences of input ids (or a batch of pairs) is provided with *truncation_strategy = longest_first* or *True*, an error is raised instead of returning overflowing tokens.
- `return_special_tokens_mask` (*bool*, *optional*, defaults to *False*) – Whether or not to return special tokens mask information.
- `return_offsets_mapping` (*bool*, *optional*, defaults to *False*) – Whether or not to return *(char_start, char_end)* for each token.

  This is only available on fast tokenizers inheriting from [*PreTrainedTokenizerFast*], if using Python's tokenizer, this method will raise *NotImplementedError*.
- `return_length` (*bool*, *optional*, defaults to *False*) – Whether or not to return the lengths of the encoded inputs.
- `verbose` (*bool*, *optional*, defaults to *True*) – Whether or not to print more information and warnings.
- `**kwargs` – passed to the *self.tokenize()* method

Returns

A [*BatchEncoding*] with the following fields:

- **input_ids** – List of token ids to be fed to a model.

  [What are input IDs?](../glossary#input-ids)
- **token_type_ids** – List of token type ids to be fed to a model (when *return_token_type_ids=True* or if *"token_type_ids"* is in *self.model_input_names*).

  [What are token type IDs?](../glossary#token-type-ids)
- **attention_mask** – List of indices specifying which tokens should be attended to by the model (when *return_attention_mask=True* or if *"attention_mask"* is in *self.model_input_names*).

  [What are attention masks?](../glossary#attention-mask)

- **overflowing_tokens** – List of overflowing tokens sequences (when a *max_length* is specified and *return_overflowing_tokens=True*).
- **num_truncated_tokens** – Number of tokens truncated (when a *max_length* is specified and *return_overflowing_tokens=True*).
- **special_tokens_mask** – List of 0s and 1s, with 1 specifying added special tokens and 0 specifying regular sequence tokens (when *add_special_tokens=True* and *return_special_tokens_mask=True*).
- **length** – The length of the inputs (when *return_length=True*)

Return type
[*BatchEncoding*]

`property bos_token: str`

Beginning of sentence token. Log an error if used while not having been set.

Type
str

`property bos_token_id: int | None`

Id of the beginning of sentence token in the vocabulary. Returns *None* if the token has not been set.

Type
Optional[int]

`build_inputs_with_special_tokens`(*token_ids_0*, *token_ids_1=None*)

Build model inputs from a sequence or a pair of sequence for sequence classification tasks by concatenating and adding special tokens.

This implementation does not add special tokens and this method should be overridden in a subclass.

Parameters

- `token_ids_0` (*List[int]*) – The first tokenized sequence.
- `token_ids_1` (*List[int]*, *optional*) – The second tokenized sequence.

Returns
The model input with special tokens.

Return type
List[int]

`property can_save_slow_tokenizer: bool`

Whether or not the slow tokenizer can be saved. Usually for sentencepiece based slow tokenizer, this can only be *True* if the original *"sentencepiece.model"* was not deleted.

Type
bool

`static clean_up_tokenization`(*out_string: str*) → str

Clean up a list of simple English tokenization artifacts like spaces before punctuations and abbreviated forms.

Parameters
`out_string` (*str*) – The text to clean up.

Returns
The cleaned-up string.

Return type
str

property cls_token: str

Classification token, to extract a summary of an input sequence leveraging self-attention along the full depth of the model. Log an error if used while not having been set.

Type
str

property cls_token_id: int | None

Id of the classification token in the vocabulary, to extract a summary of an input sequence leveraging self-attention along the full depth of the model.

Returns *None* if the token has not been set.

Type
Optional[int]

convert_ids_to_tokens(*ids: int* | *List[int]*, *skip_special_tokens: bool* = *False*) → str | List[str]

Converts a single index or a sequence of indices in a token or a sequence of tokens, using the vocabulary and added tokens.

Parameters

- **ids** (*int* or *List[int]*) – The token id (or token ids) to convert to tokens.
- **skip_special_tokens** (*bool*, *optional*, defaults to *False*) – Whether or not to remove special tokens in the decoding.

Returns
The decoded token(s).

Return type
str or *List[str]*

convert_tokens_to_ids(*tokens: str* | *List[str]*) → int | List[int]

Converts a token string (or a sequence of tokens) in a single integer id (or a sequence of ids), using the vocabulary.

Parameters
tokens (*str* or *List[str]*) – One or several token(s) to convert to token id(s).

Returns
The token id or list of token ids.

Return type
int or *List[int]*

convert_tokens_to_string(*tokens: List[str]*) → str

Converts a sequence of tokens in a single string. The most simple way to do it is *" ".join(tokens)* but we often want to remove sub-word tokenization artifacts at the same time.

Parameters
tokens (*List[str]*) – The token to join in a string.

Returns
The joined tokens.

Return type
str

`create_token_type_ids_from_sequences`(*token_ids_0: List[int]*, *token_ids_1: List[int] | None = None*) → List[int]

Create a mask from the two sequences passed to be used in a sequence-pair classification task. RoBERTa does not make use of token type ids, therefore a list of zeros is returned.

Parameters

- `token_ids_0` (*List[int]*) – List of IDs.
- `token_ids_1` (*List[int]*, *optional*) – Optional second list of IDs for sequence pairs.

Returns
List of zeros.

Return type
List[int]

`decode`(*token_ids: int | List[int] | np.ndarray | torch.Tensor | tf.Tensor*, *skip_special_tokens: bool = False*, *clean_up_tokenization_spaces: bool = None*, ***kwargs*) → str

Converts a sequence of ids in a string, using the tokenizer and vocabulary with options to remove special tokens and clean up tokenization spaces.

Similar to doing *self.convert_tokens_to_string(self.convert_ids_to_tokens(token_ids))*.

Parameters

- `token_ids` (*Union[int, List[int], np.ndarray, torch.Tensor, tf.Tensor]*) – List of tokenized input ids. Can be obtained using the *__call__* method.
- `skip_special_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not to remove special tokens in the decoding.
- `clean_up_tokenization_spaces` (*bool*, *optional*) – Whether or not to clean up the tokenization spaces. If *None*, will default to *self.clean_up_tokenization_spaces*.
- `kwargs` (additional keyword arguments, *optional*) – Will be passed to the underlying model specific decode method.

Returns
The decoded sentence.

Return type
str

`property decoder: Decoder`

The Rust decoder for this tokenizer.

Type
tokenizers.decoders.Decoder

`property default_chat_template`

This template formats inputs in the standard ChatML format. See https://github.com/openai/openai-python/blob/main/chatml.md

`encode`(*text: str | List[str] | List[int]*, *text_pair: str | List[str] | List[int] | None = None*, *add_special_tokens: bool = True*, *padding: bool | str | PaddingStrategy = False*, *truncation: bool | str | TruncationStrategy | None = None*, *max_length: int | None = None*, *stride: int = 0*, *return_tensors: str | TensorType | None = None*, ***kwargs*) → List[int]

Converts a string to a sequence of ids (integer), using the tokenizer and vocabulary.

Same as doing *self.convert_tokens_to_ids(self.tokenize(text))*.

Parameters

- **text** (*str*, *List[str]* or *List[int]*) – The first sequence to be encoded. This can be a string, a list of strings (tokenized string using the *tokenize* method) or a list of integers (tokenized string ids using the *convert_tokens_to_ids* method).
- **text_pair** (*str*, *List[str]* or *List[int]*, *optional*) – Optional second sequence to be encoded. This can be a string, a list of strings (tokenized string using the *tokenize* method) or a list of integers (tokenized string ids using the *convert_tokens_to_ids* method).
- **add_special_tokens** (*bool*, *optional*, defaults to *True*) – Whether or not to add special tokens when encoding the sequences. This will use the underlying *PretrainedTokenizerBase.build_inputs_with_special_tokens* function, which defines which tokens are automatically added to the input ids. This is usefull if you want to add *bos* or *eos* tokens automatically.
- **padding** (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *False*) – Activates and controls padding. Accepts the following values:
  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).
  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).
- **truncation** (*bool*, *str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *False*) – Activates and controls truncation. Accepts the following values:
  - *True* or *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *False* or *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- **max_length** (*int*, *optional*) – Controls the maximum length to use by one of the truncation/padding parameters.

  If left unset or set to *None*, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.
- **stride** (*int*, *optional*, defaults to 0) – If set to a number along with *max_length*, the overflowing tokens returned when *return_overflowing_tokens=True* will contain some tokens from the end of the truncated sequence returned to provide some overlap between truncated and overflowing sequences. The value of this argument defines the number of overlapping tokens.

- **is_split_into_words** (*bool*, *optional*, defaults to *False*) – Whether or not the input is already pre-tokenized (e.g., split into words). If set to *True*, the tokenizer assumes the input is already split into words (for instance, by splitting it on whitespace) which it will tokenize. This is useful for NER or token classification.
- **pad_to_multiple_of** (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value. Requires *padding* to be activated. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= *7.5* (Volta).
- **return_tensors** (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:
  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.
- ****kwargs** – Passed along to the *.tokenize()* method.

Returns
The tokenized ids of the text.

Return type
List[int], *torch.Tensor*, *tf.Tensor* or *np.ndarray*

encode_plus(*text: str | List[str] | List[int]*, *text_pair: str | List[str] | List[int] | None = None*, *add_special_tokens: bool = True*, *padding: bool | str | PaddingStrategy = False*, *truncation: bool | str | TruncationStrategy | None = None*, *max_length: int | None = None*, *stride: int = 0*, *is_split_into_words: bool = False*, *pad_to_multiple_of: int | None = None*, *return_tensors: str | TensorType | None = None*, *return_token_type_ids: bool | None = None*, *return_attention_mask: bool | None = None*, *return_overflowing_tokens: bool = False*, *return_special_tokens_mask: bool = False*, *return_offsets_mapping: bool = False*, *return_length: bool = False*, *verbose: bool = True*, ***kwargs*) → BatchEncoding

Tokenize and prepare for the model a sequence or a pair of sequences.

<Tip warning={true}>

This method is deprecated, *__call__* should be used instead.

</Tip>

Parameters

- **text** (*str*, *List[str]* or *List[int]* (the latter only for not-fast tokenizers)) – The first sequence to be encoded. This can be a string, a list of strings (tokenized string using the *tokenize* method) or a list of integers (tokenized string ids using the *convert_tokens_to_ids* method).
- **text_pair** (*str*, *List[str]* or *List[int]*, *optional*) – Optional second sequence to be encoded. This can be a string, a list of strings (tokenized string using the *tokenize* method) or a list of integers (tokenized string ids using the *convert_tokens_to_ids* method).
- **add_special_tokens** (*bool*, *optional*, defaults to *True*) – Whether or not to add special tokens when encoding the sequences. This will use the underlying *PretrainedTokenizerBase.build_inputs_with_special_tokens* function, which defines which tokens are automatically added to the input ids. This is usefull if you want to add *bos* or *eos* tokens automatically.
- **padding** (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *False*) – Activates and controls padding. Accepts the following values:
  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).

  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).

- **truncation** (*bool*, *str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *False*) – Activates and controls truncation. Accepts the following values:
  - *True* or *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *False* or *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- **max_length** (*int*, *optional*) – Controls the maximum length to use by one of the truncation/padding parameters.

  If left unset or set to *None*, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.
- **stride** (*int*, *optional*, defaults to 0) – If set to a number along with *max_length*, the overflowing tokens returned when *return_overflowing_tokens=True* will contain some tokens from the end of the truncated sequence returned to provide some overlap between truncated and overflowing sequences. The value of this argument defines the number of overlapping tokens.
- **is_split_into_words** (*bool*, *optional*, defaults to *False*) – Whether or not the input is already pre-tokenized (e.g., split into words). If set to *True*, the tokenizer assumes the input is already split into words (for instance, by splitting it on whitespace) which it will tokenize. This is useful for NER or token classification.
- **pad_to_multiple_of** (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value. Requires *padding* to be activated. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability *>= 7.5* (Volta).
- **return_tensors** (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:
  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.
- **return_token_type_ids** (*bool*, *optional*) – Whether to return token type IDs. If left to the default, will return the token type IDs according to the specific tokenizer's default, defined by the *return_outputs* attribute.

[What are token type IDs?](../glossary#token-type-ids)

- `return_attention_mask` (*bool*, *optional*) – Whether to return the attention mask. If left to the default, will return the attention mask according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are attention masks?](../glossary#attention-mask)

- `return_overflowing_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not to return overflowing token sequences. If a pair of sequences of input ids (or a batch of pairs) is provided with *truncation_strategy = longest_first* or *True*, an error is raised instead of returning overflowing tokens.

- `return_special_tokens_mask` (*bool*, *optional*, defaults to *False*) – Whether or not to return special tokens mask information.

- `return_offsets_mapping` (*bool*, *optional*, defaults to *False*) – Whether or not to return *(char_start, char_end)* for each token.

  This is only available on fast tokenizers inheriting from [*PreTrainedTokenizerFast*], if using Python's tokenizer, this method will raise *NotImplementedError*.

- `return_length` (*bool*, *optional*, defaults to *False*) – Whether or not to return the lengths of the encoded inputs.

- `verbose` (*bool*, *optional*, defaults to *True*) – Whether or not to print more information and warnings.

- `**kwargs` – passed to the *self.tokenize()* method

Returns

A [*BatchEncoding*] with the following fields:

- **input_ids** – List of token ids to be fed to a model.

  [What are input IDs?](../glossary#input-ids)

- **token_type_ids** – List of token type ids to be fed to a model (when *return_token_type_ids=True* or if *"token_type_ids"* is in *self.model_input_names*).

  [What are token type IDs?](../glossary#token-type-ids)

- **attention_mask** – List of indices specifying which tokens should be attended to by the model (when *return_attention_mask=True* or if *"attention_mask"* is in *self.model_input_names*).

  [What are attention masks?](../glossary#attention-mask)

- **overflowing_tokens** – List of overflowing tokens sequences (when a *max_length* is specified and *return_overflowing_tokens=True*).

- **num_truncated_tokens** – Number of tokens truncated (when a *max_length* is specified and *return_overflowing_tokens=True*).

- **special_tokens_mask** – List of 0s and 1s, with 1 specifying added special tokens and 0 specifying regular sequence tokens (when *add_special_tokens=True* and *return_special_tokens_mask=True*).

- **length** – The length of the inputs (when *return_length=True*)

Return type

[*BatchEncoding*]

`property eos_token: str`

End of sentence token. Log an error if used while not having been set.

Type
str

`property eos_token_id: int | None`

Id of the end of sentence token in the vocabulary. Returns *None* if the token has not been set.

Type
Optional[int]

`featurize`(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- `datapoints` (`Iterable[Any]`) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- `log_every_n` (`int, default 1000`) – Logs featurization progress every *log_every_n* steps.

Returns
A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

`classmethod from_pretrained`(*pretrained_model_name_or_path: str | PathLike*, **init_inputs*, *cache_dir: str | PathLike | None = None*, *force_download: bool = False*, *local_files_only: bool = False*, *token: bool | str | None = None*, *revision: str = 'main'*, *trust_remote_code=False*, ***kwargs*)

Instantiate a [*~tokenization_utils_base.PreTrainedTokenizerBase*] (or a derived class) from a predefined tokenizer.

Parameters

- `pretrained_model_name_or_path` (*str* or *os.PathLike*) – Can be either:
  - A string, the *model id* of a predefined tokenizer hosted inside a model repo on huggingface.co.
  - A path to a *directory* containing vocabulary files required by the tokenizer, for instance saved using the [*~tokenization_utils_base.PreTrainedTokenizerBase.save_pretrained*] method, e.g., *./my_model_directory/*.
  - (**Deprecated**, not applicable to all derived classes) A path or url to a single saved vocabulary file (if and only if the tokenizer only requires a single vocabulary file like Bert or XLNet), e.g., *./my_model_directory/vocab.txt*.
- `cache_dir` (*str* or *os.PathLike*, *optional*) – Path to a directory in which a downloaded predefined tokenizer vocabulary files should be cached if the standard cache should not be used.
- `force_download` (*bool*, *optional*, defaults to *False*) – Whether or not to force the (re-)download the vocabulary files and override the cached versions if they exist.
- `resume_download` (*bool*, *optional*, defaults to *False*) – Whether or not to delete incompletely received files. Attempt to resume the download if such a file exists.

- **proxies** (*Dict[str, str]*, *optional*) – A dictionary of proxy servers to use by protocol or endpoint, e.g., *{'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}*. The proxies are used on each request.
- **token** (*str* or *bool*, *optional*) – The token to use as HTTP bearer authorization for remote files. If *True*, will use the token generated when running *huggingface-cli login* (stored in *~/.huggingface*).
- **local_files_only** (*bool*, *optional*, defaults to *False*) – Whether or not to only rely on local files and not to attempt to download any files.
- **revision** (*str*, *optional*, defaults to *"main"*) – The specific model version to use. It can be a branch name, a tag name, or a commit id, since we use a git-based system for storing models and other artifacts on huggingface.co, so *revision* can be any identifier allowed by git.
- **subfolder** (*str*, *optional*) – In case the relevant files are located inside a subfolder of the model repo on huggingface.co (e.g. for facebook/rag-token-base), specify it here.
- **inputs** (additional positional arguments, *optional*) – Will be passed along to the Tokenizer *__init__* method.
- **trust_remote_code** (*bool*, *optional*, defaults to *False*) – Whether or not to allow for custom models defined on the Hub in their own modeling files. This option should only be set to *True* for repositories you trust and in which you have read the code, as it will execute code present on the Hub on your local machine.
- **kwargs** (additional keyword arguments, *optional*) – Will be passed to the Tokenizer *__init__* method. Can be used to set special tokens like *bos_token*, *eos_token*, *unk_token*, *sep_token*, *pad_token*, *cls_token*, *mask_token*, *additional_special_tokens*. See parameters in the *__init__* for more details.

<Tip>

Passing *token=True* is required when you want to use a private model.

</Tip>

Examples:

```python # We can't instantiate directly the base class *PreTrainedTokenizerBase* so let's show our examples on a derived class: BertTokenizer # Download vocabulary from huggingface.co and cache. tokenizer = BertTokenizer.from_pretrained("google-bert/bert-base-uncased")

# Download vocabulary from huggingface.co (user-uploaded) and cache. tokenizer = BertTokenizer.from_pretrained("dbmdz/bert-base-german-cased")

# If vocabulary files are in a directory (e.g. tokenizer was saved using *save_pretrained('./test/saved_model/')*) tokenizer = BertTokenizer.from_pretrained("./test/saved_model/")

# If the tokenizer uses a single vocabulary file, you can point directly to this file tokenizer = BertTokenizer.from_pretrained("./test/saved_model/my_vocab.txt")

# You can link tokens to special vocabulary when instantiating tokenizer = BertTokenizer.from_pretrained("google-bert/bert-base-uncased", unk_token="<unk>") # You should be sure '<unk>' is in the vocabulary when doing that. # Otherwise use tokenizer.add_special_tokens({'unk_token': '<unk>'}) instead) assert tokenizer.unk_token == "<unk>" ```

**get_added_vocab()** → Dict[str, int]

Returns the added tokens in the vocabulary as a dictionary of token to index.
```

Returns
The added tokens.

Return type
Dict[str, int]

`get_special_tokens_mask`(*token_ids_0: List[int]*, *token_ids_1: List[int] | None = None*, *already_has_special_tokens: bool = False*) → List[int]

Retrieves sequence ids from a token list that has no special tokens added. This method is called when adding special tokens using the tokenizer *prepare_for_model* or *encode_plus* methods.

Parameters

- `token_ids_0` (*List[int]*) – List of ids of the first sequence.
- `token_ids_1` (*List[int]*, *optional*) – List of ids of the second sequence.
- `already_has_special_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not the token list is already formatted with special tokens for the model.

Returns
1 for a special token, 0 for a sequence token.

Return type
A list of integers in the range [0, 1]

`get_vocab`() → Dict[str, int]

Returns the vocabulary as a dictionary of token to index.

tokenizer.get_vocab()[token] is equivalent to *tokenizer.convert_tokens_to_ids(token)* when *token* is in the vocab.

Returns
The vocabulary.

Return type
Dict[str, int]

`property mask_token: str`

Mask token, to use when training a model with masked-language modeling. Log an error if used while not having been set.

Roberta tokenizer has a special mask token to be usable in the fill-mask pipeline. The mask token will greedily comprise the space before the *<mask>*.

Type
str

`property mask_token_id: int | None`

Id of the mask token in the vocabulary, used when training a model with masked-language modeling. Returns *None* if the token has not been set.

Type
Optional[int]

`property max_len_sentences_pair: int`

The maximum combined length of a pair of sentences that can be fed to the model.

Type
int

`property max_len_single_sentence: int`

The maximum length of a sentence that can be fed to the model.

Type
int

`num_special_tokens_to_add`(*pair: bool = False*) → int

Returns the number of added tokens when encoding a sequence with special tokens.

<Tip>

This encodes a dummy input and checks the number of added tokens, and is therefore not efficient. Do not put this inside your training loop.

</Tip>

Parameters
`pair` (*bool*, *optional*, defaults to *False*) – Whether the number of added tokens should be computed in the case of a sequence pair or a single sequence.

Returns
Number of special tokens added to sequences.

Return type
int

`pad`(*encoded_inputs: BatchEncoding | List[BatchEncoding] | Dict[str, List[int]] | Dict[str, List[List[int]]] | List[Dict[str, List[int]]]*, *padding: bool | str | PaddingStrategy = True*, *max_length: int | None = None*, *pad_to_multiple_of: int | None = None*, *return_attention_mask: bool | None = None*, *return_tensors: str | TensorType | None = None*, *verbose: bool = True*) → BatchEncoding

Pad a single encoded input or a batch of encoded inputs up to predefined length or to the max sequence length in the batch.

Padding side (left/right) padding token ids are defined at the tokenizer level (with *self.padding_side*, *self.pad_token_id* and *self.pad_token_type_id*).

Please note that with a fast tokenizer, using the *__call__* method is faster than using a method to encode the text followed by a call to the *pad* method to get a padded encoding.

<Tip>

If the *encoded_inputs* passed are dictionary of numpy arrays, PyTorch tensors or TensorFlow tensors, the result will use the same type unless you provide a different tensor type with *return_tensors*. In the case of PyTorch tensors, you will lose the specific device of your tensors however.

</Tip>

Parameters

- `encoded_inputs` ([*BatchEncoding*], list of [*BatchEncoding*], *Dict[str, List[int]]*, *Dict[str, List[List[int]]* or *List[Dict[str, List[int]]]*) – Tokenized inputs. Can represent one input ([*BatchEncoding*] or *Dict[str, List[int]]*) or a batch of tokenized inputs (list of [*BatchEncoding*], *Dict[str, List[List[int]]]* or *List[Dict[str, List[int]]]*) so you can use this method during preprocessing as well as in a PyTorch Dataloader collate function.

  Instead of *List[int]* you can have tensors (numpy arrays, PyTorch tensors or TensorFlow tensors), see the note above for the return type.

- `padding` (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *True*) –

  Select a strategy to pad the returned sequences (according to the model's padding side and padding
  index) among:

  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).
  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).

- `max_length` (*int*, *optional*) – Maximum length of the returned list and optionally padding length (see above).
- `pad_to_multiple_of` (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value.

  This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= *7.5* (Volta).
- `return_attention_mask` (*bool*, *optional*) – Whether to return the attention mask. If left to the default, will return the attention mask according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are attention masks?](../glossary#attention-mask)
- `return_tensors` (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:
  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.
- `verbose` (*bool*, *optional*, defaults to *True*) – Whether or not to print more information and warnings.

`property pad_token: str`

Padding token. Log an error if used while not having been set.

Type
str

`property pad_token_id: int | None`

Id of the padding token in the vocabulary. Returns *None* if the token has not been set.

Type
Optional[int]

`property pad_token_type_id: int`

Id of the padding token type in the vocabulary.

Type
int

`prepare_for_model`(*ids: List[int]*, *pair_ids: List[int] | None = None*, *add_special_tokens: bool = True*, *padding: bool | str | PaddingStrategy = False*, *truncation: bool | str | TruncationStrategy | None = None*, *max_length: int | None = None*, *stride: int = 0*, *pad_to_multiple_of: int | None = None*, *return_tensors: str | TensorType | None = None*, *return_token_type_ids: bool | None = None*, *return_attention_mask: bool | None = None*, *return_overflowing_tokens: bool = False*, *return_special_tokens_mask: bool = False*, *return_offsets_mapping: bool = False*, *return_length: bool = False*, *verbose: bool = True*, *prepend_batch_axis: bool = False*, ***kwargs*) → BatchEncoding

Prepares a sequence of input id, or a pair of sequences of inputs ids so that it can be used by the model. It adds special tokens, truncates sequences if overflowing while taking into account the special tokens and manages a moving window (with user defined stride) for overflowing tokens. Please Note, for *pair_ids* different than *None* and *truncation_strategy = longest_first* or *True*, it is not possible to return overflowing tokens. Such a combination of arguments will raise an error.

Parameters

- `ids` (*List[int]*) – Tokenized input ids of the first sequence. Can be obtained from a string by chaining the *tokenize* and *convert_tokens_to_ids* methods.
- `pair_ids` (*List[int]*, *optional*) – Tokenized input ids of the second sequence. Can be obtained from a string by chaining the *tokenize* and *convert_tokens_to_ids* methods.
- `add_special_tokens` (*bool*, *optional*, defaults to *True*) – Whether or not to add special tokens when encoding the sequences. This will use the underlying *PretrainedTokenizer-Base.build_inputs_with_special_tokens* function, which defines which tokens are automatically added to the input ids. This is usefull if you want to add *bos* or *eos* tokens automatically.
- `padding` (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *False*) – Activates and controls padding. Accepts the following values:
  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).
  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).
- `truncation` (*bool*, *str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *False*) – Activates and controls truncation. Accepts the following values:
  - *True* or *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *False* or *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- `max_length` (*int*, *optional*) – Controls the maximum length to use by one of the truncation/padding parameters.

  If left unset or set to *None*, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.

- **stride** (*int*, *optional*, defaults to 0) – If set to a number along with *max_length*, the overflowing tokens returned when *return_overflowing_tokens=True* will contain some tokens from the end of the truncated sequence returned to provide some overlap between truncated and overflowing sequences. The value of this argument defines the number of overlapping tokens.
- **is_split_into_words** (*bool*, *optional*, defaults to *False*) – Whether or not the input is already pre-tokenized (e.g., split into words). If set to *True*, the tokenizer assumes the input is already split into words (for instance, by splitting it on whitespace) which it will tokenize. This is useful for NER or token classification.
- **pad_to_multiple_of** (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value. Requires *padding* to be activated. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= *7.5* (Volta).
- **return_tensors** (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:
  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.
- **return_token_type_ids** (*bool*, *optional*) – Whether to return token type IDs. If left to the default, will return the token type IDs according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are token type IDs?](../glossary#token-type-ids)
- **return_attention_mask** (*bool*, *optional*) – Whether to return the attention mask. If left to the default, will return the attention mask according to the specific tokenizer's default, defined by the *return_outputs* attribute.

  [What are attention masks?](../glossary#attention-mask)
- **return_overflowing_tokens** (*bool*, *optional*, defaults to *False*) – Whether or not to return overflowing token sequences. If a pair of sequences of input ids (or a batch of pairs) is provided with *truncation_strategy = longest_first* or *True*, an error is raised instead of returning overflowing tokens.
- **return_special_tokens_mask** (*bool*, *optional*, defaults to *False*) – Whether or not to return special tokens mask information.
- **return_offsets_mapping** (*bool*, *optional*, defaults to *False*) – Whether or not to return *(char_start, char_end)* for each token.

  This is only available on fast tokenizers inheriting from [*PreTrainedTokenizerFast*], if using Python's tokenizer, this method will raise *NotImplementedError*.
- **return_length** (*bool*, *optional*, defaults to *False*) – Whether or not to return the lengths of the encoded inputs.
- **verbose** (*bool*, *optional*, defaults to *True*) – Whether or not to print more information and warnings.
- ****kwargs** – passed to the *self.tokenize()* method

Returns

A [*BatchEncoding*] with the following fields:

- **input_ids** – List of token ids to be fed to a model.

[What are input IDs?](../glossary#input-ids)

- **token_type_ids** – List of token type ids to be fed to a model (when *return_token_type_ids=True* or if *"token_type_ids"* is in *self.model_input_names*).

  [What are token type IDs?](../glossary#token-type-ids)

- **attention_mask** – List of indices specifying which tokens should be attended to by the model (when *return_attention_mask=True* or if *"attention_mask"* is in *self.model_input_names*).

  [What are attention masks?](../glossary#attention-mask)

- **overflowing_tokens** – List of overflowing tokens sequences (when a *max_length* is specified and *return_overflowing_tokens=True*).
- **num_truncated_tokens** – Number of tokens truncated (when a *max_length* is specified and *return_overflowing_tokens=True*).
- **special_tokens_mask** – List of 0s and 1s, with 1 specifying added special tokens and 0 specifying regular sequence tokens (when *add_special_tokens=True* and *return_special_tokens_mask=True*).
- **length** – The length of the inputs (when *return_length=True*)

Return type

[*BatchEncoding*]

`prepare_seq2seq_batch`(*src_texts: List[str]*, *tgt_texts: List[str] | None = None*, *max_length: int | None = None*, *max_target_length: int | None = None*, *padding: str = 'longest'*, *return_tensors: str | None = None*, *truncation: bool = True*, ***kwargs*) → BatchEncoding

Prepare model inputs for translation. For best performance, translate one sentence at a time.

Parameters

- `src_texts` (*List[str]*) – List of documents to summarize or source language texts.
- `tgt_texts` (*list*, *optional*) – List of summaries or target language texts.
- `max_length` (*int*, *optional*) – Controls the maximum length for encoder inputs (documents to summarize or source language texts) If left unset or set to *None*, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.
- `max_target_length` (*int*, *optional*) – Controls the maximum length of decoder inputs (target language texts or summaries) If left unset or set to *None*, this will use the max_length value.
- `padding` (*bool*, *str* or [*~utils.PaddingStrategy*], *optional*, defaults to *False*) – Activates and controls padding. Accepts the following values:
  - *True* or *'longest'*: Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).
  - *'max_length'*: Pad to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided.
  - *False* or *'do_not_pad'* (default): No padding (i.e., can output a batch with sequences of different lengths).
- `return_tensors` (*str* or [*~utils.TensorType*], *optional*) – If set, will return tensors instead of list of python integers. Acceptable values are:

  - *'tf'*: Return TensorFlow *tf.constant* objects.
  - *'pt'*: Return PyTorch *torch.Tensor* objects.
  - *'np'*: Return Numpy *np.ndarray* objects.

- `truncation` (*bool*, *str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *True*) – Activates and controls truncation. Accepts the following values:
  - *True* or *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *False* or *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- `**kwargs` – Additional keyword arguments passed along to *self.__call__*.

Returns

A [*BatchEncoding*] with the following fields:

- **input_ids** – List of token ids to be fed to the encoder.
- **attention_mask** – List of indices specifying which tokens should be attended to by the model.
- **labels** – List of token ids for tgt_texts.

The full set of keys *[input_ids, attention_mask, labels]*, will only be returned if tgt_texts is passed. Otherwise, input_ids, attention_mask will be the only keys.

Return type

[*BatchEncoding*]

`push_to_hub`(*repo_id: str*, *use_temp_dir: bool | None = None*, *commit_message: str | None = None*, *private: bool | None = None*, *token: bool | str | None = None*, *max_shard_size: int | str | None = '5GB'*, *create_pr: bool = False*, *safe_serialization: bool = True*, *revision: str | None = None*, *commit_description: str | None = None*, *tags: List[str] | None = None*, ***deprecated_kwargs*) → str

Upload the tokenizer files to the Model Hub.

Parameters

- `repo_id` (*str*) – The name of the repository you want to push your tokenizer to. It should contain your organization name when pushing to a given organization.
- `use_temp_dir` (*bool*, *optional*) – Whether or not to use a temporary directory to store the files saved before they are pushed to the Hub. Will default to *True* if there is no directory named like *repo_id*, *False* otherwise.
- `commit_message` (*str*, *optional*) – Message to commit while pushing. Will default to *"Upload tokenizer"*.

- `private` (*bool*, *optional*) – Whether or not the repository created should be private.
- `token` (*bool* or *str*, *optional*) – The token to use as HTTP bearer authorization for remote files. If *True*, will use the token generated when running *huggingface-cli login* (stored in *~/.huggingface*). Will default to *True* if *repo_url* is not specified.
- `max_shard_size` (*int* or *str*, *optional*, defaults to *"5GB"*) – Only applicable for models. The maximum size for a checkpoint before being sharded. Checkpoints shard will then be each of size lower than this size. If expressed as a string, needs to be digits followed by a unit (like *"5MB"*). We default it to *"5GB"* so that users can easily load models on free-tier Google Colab instances without any CPU OOM issues.
- `create_pr` (*bool*, *optional*, defaults to *False*) – Whether or not to create a PR with the uploaded files or directly commit.
- `safe_serialization` (*bool*, *optional*, defaults to *True*) – Whether or not to convert the model weights in safetensors format for safer serialization.
- `revision` (*str*, *optional*) – Branch to push the uploaded files to.
- `commit_description` (*str*, *optional*) – The description of the commit that will be created
- `tags` (*List[str]*, *optional*) – List of tags to push on the Hub.

Examples:

```python from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-cased")

# Push the tokenizer to your namespace with the name "my-finetuned-bert". tokenizer.push_to_hub("my-finetuned-bert")

# Push the tokenizer to an organization with the name "my-finetuned-bert". tokenizer.push_to_hub("huggingface/my-finetuned-bert") ```

**classmethod register_for_auto_class**(*auto_class='AutoTokenizer'*)

Register this class with a given auto class. This should only be used for custom tokenizers as the ones in the library are already mapped with *AutoTokenizer*.

<Tip warning={true}>

This API is experimental and may have some slight breaking changes in the next releases.

</Tip>

**Parameters**
`auto_class` (*str* or *type*, *optional*, defaults to *"AutoTokenizer"*) – The auto class to register this new tokenizer with.

**sanitize_special_tokens**() → int

The *sanitize_special_tokens* is now deprecated kept for backward compatibility and will be removed in transformers v5.

**save_pretrained**(*save_directory: str | PathLike*, *legacy_format: bool | None = None*, *filename_prefix: str | None = None*, *push_to_hub: bool = False*, ***kwargs*) → Tuple[str]

Save the full tokenizer state.

This method make sure the full tokenizer can then be re-loaded using the [*~tokenization_utils_base.PreTrainedTokenizer.from_pretrained*] class method..

Warning,None This won't save modifications you may have applied to the tokenizer after the instantiation (for instance, modifying *tokenizer.do_lower_case* after creation).
```

Parameters

- `save_directory` (*str* or *os.PathLike*) – The path to a directory where the tokenizer will be saved.
- `legacy_format` (*bool*, *optional*) – Only applicable for a fast tokenizer. If unset (default), will save the tokenizer in the unified JSON format as well as in legacy format if it exists, i.e. with tokenizer specific vocabulary and a separate added_tokens files.

  If *False*, will only save the tokenizer in the unified JSON format. This format is incompatible with "slow" tokenizers (not powered by the *tokenizers* library), so the tokenizer will not be able to be loaded in the corresponding "slow" tokenizer.

  If *True*, will save the tokenizer in legacy format. If the "slow" tokenizer doesn't exits, a value error is raised.
- `filename_prefix` (*str*, *optional*) – A prefix to add to the names of the files saved by the tokenizer.
- `push_to_hub` (*bool*, *optional*, defaults to *False*) – Whether or not to push your model to the Hugging Face model hub after saving it. You can specify the repository you want to push to with *repo_id* (will default to the name of *save_directory* in your namespace).
- `kwargs` (*Dict[str, Any]*, *optional*) – Additional key word arguments passed along to the [*~utils.PushToHubMixin.push_to_hub*] method.

Returns
The files saved.

Return type
A tuple of *str*

`save_vocabulary`(*save_directory: str*, *filename_prefix: str | None = None*) → Tuple[str]

Save only the vocabulary of the tokenizer (vocabulary + added tokens).

This method won't save the configuration and special token mappings of the tokenizer. Use [*~PreTrainedTokenizerFast._save_pretrained*] to save the whole state of the tokenizer.

Parameters

- `save_directory` (*str*) – The directory in which to save the vocabulary.
- `filename_prefix` (*str*, *optional*) – An optional prefix to add to the named of the saved files.

Returns
Paths to the files saved.

Return type
Tuple(str)

`property sep_token: str`

Separation token, to separate context and query in an input sequence. Log an error if used while not having been set.

Type
str

`property sep_token_id: int | None`

Id of the separation token in the vocabulary, to separate context and query in an input sequence. Returns *None* if the token has not been set.

Type
Optional[int]

`set_truncation_and_padding`(*padding_strategy: PaddingStrategy*, *truncation_strategy: TruncationStrategy*, *max_length: int*, *stride: int*, *pad_to_multiple_of: int | None*)

Define the truncation and the padding strategies for fast tokenizers (provided by HuggingFace tokenizers library) and restore the tokenizer settings afterwards.

The provided tokenizer has no padding / truncation strategy before the managed section. If your tokenizer set a padding / truncation strategy before, then it will be reset to no padding / truncation when exiting the managed section.

Parameters

- `padding_strategy` ([*~utils.PaddingStrategy*]) – The kind of padding that will be applied to the input
- `truncation_strategy` ([*~tokenization_utils_base.TruncationStrategy*]) – The kind of truncation that will be applied to the input
- `max_length` (*int*) – The maximum size of a sequence.
- `stride` (*int*) – The stride to use when handling overflow.
- `pad_to_multiple_of` (*int*, *optional*) – If set will pad the sequence to a multiple of the provided value. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= *7.5* (Volta).

`slow_tokenizer_class`

alias of `RobertaTokenizer`

`property special_tokens_map: Dict[str, str | List[str]]`

A dictionary mapping special token class attributes (*cls_token*, *unk_token*, etc.) to their values (*'<unk>'*, *'<cls>'*, etc.).

Convert potential tokens of *tokenizers.AddedToken* type to string.

Type
Dict[str, Union[str, List[str]]]

`property special_tokens_map_extended: Dict[str, str | AddedToken | List[str | AddedToken]]`

A dictionary mapping special token class attributes (*cls_token*, *unk_token*, etc.) to their values (*'<unk>'*, *'<cls>'*, etc.).

Don't convert tokens of *tokenizers.AddedToken* type to string so they can be used to control more finely how special tokens are tokenized.

Type
Dict[str, Union[str, tokenizers.AddedToken, List[Union[str, tokenizers.AddedToken]]]]

`tokenize`(*text: str*, *pair: str | None = None*, *add_special_tokens: bool = False*, ***kwargs*) → List[str]

Converts a string into a sequence of tokens, replacing unknown tokens with the *unk_token*.

Parameters

- `text` (*str*) – The sequence to be encoded.
- `pair` (*str*, *optional*) – A second sequence to be encoded with the first.
- `add_special_tokens` (*bool*, *optional*, defaults to *False*) – Whether or not to add the special tokens associated with the corresponding model.

- **kwargs** (additional keyword arguments, *optional*) – Will be passed to the underlying model specific encode method. See details in [*~PreTrainedTokenizerBase.__call__*]

Returns
The list of tokens.

Return type
List[str]

`train_new_from_iterator`(*text_iterator*, *vocab_size*, *length=None*, *new_special_tokens=None*, *special_tokens_map=None*, ***kwargs*)

Trains a tokenizer on a new corpus with the same defaults (in terms of special tokens or tokenization pipeline) as the current one.

Parameters

- **text_iterator** (generator of *List[str]*) – The training corpus. Should be a generator of batches of texts, for instance a list of lists of texts if you have everything in memory.
- **vocab_size** (*int*) – The size of the vocabulary you want for your tokenizer.
- **length** (*int*, *optional*) – The total number of sequences in the iterator. This is used to provide meaningful progress tracking
- **new_special_tokens** (list of *str* or *AddedToken*, *optional*) – A list of new special tokens to add to the tokenizer you are training.
- **special_tokens_map** (*Dict[str, str]*, *optional*) – If you want to rename some of the special tokens this tokenizer uses, pass along a mapping old special token name to new special token name in this argument.
- **kwargs** (*Dict[str, Any]*, *optional*) – Additional keyword arguments passed along to the trainer from the Tokenizers library.

Returns
A new tokenizer of the same type as the original one, trained on *text_iterator*.

Return type
[*PreTrainedTokenizerFast*]

`truncate_sequences`(*ids: List[int]*, *pair_ids: List[int] | None = None*, *num_tokens_to_remove: int = 0*, *truncation_strategy: str | TruncationStrategy = 'longest_first'*, *stride: int = 0*) → Tuple[List[int], List[int], List[int]]

Truncates a sequence pair in-place following the strategy.

Parameters

- **ids** (*List[int]*) – Tokenized input ids of the first sequence. Can be obtained from a string by chaining the *tokenize* and *convert_tokens_to_ids* methods.
- **pair_ids** (*List[int]*, *optional*) – Tokenized input ids of the second sequence. Can be obtained from a string by chaining the *tokenize* and *convert_tokens_to_ids* methods.
- **num_tokens_to_remove** (*int*, *optional*, defaults to 0) – Number of tokens to remove using the truncation strategy.
- **truncation_strategy** (*str* or [*~tokenization_utils_base.TruncationStrategy*], *optional*, defaults to *False*) – The strategy to follow for truncation. Can be:
  - *'longest_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.

  - *'only_first'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'only_second'*: Truncate to a maximum length specified with the argument *max_length* or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.
  - *'do_not_truncate'* (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).
- `stride` (*int*, *optional*, defaults to 0) – If set to a positive number, the overflowing tokens returned will contain some tokens from the main sequence returned. The value of this argument defines the number of additional tokens.

Returns

The truncated *ids*, the truncated *pair_ids* and the list of overflowing tokens. Note: The *longest_first* strategy returns empty list of overflowing tokens if a pair of sequences (or a batch of pairs) is provided.

Return type

Tuple[List[int], List[int], List[int]]

`property unk_token: str`

Unknown token. Log an error if used while not having been set.

Type

str

`property unk_token_id: int | None`

Id of the unknown token in the vocabulary. Returns *None* if the token has not been set.

Type

Optional[int]

`property vocab_size: int`

Size of the base vocabulary (without the added tokens).

Type

int

RxnFeaturizer

`class RxnFeaturizer`(*tokenizer: RobertaTokenizerFast*, *sep_reagent: bool*, *max_length: int = 100*)

Reaction Featurizer.

RxnFeaturizer is a wrapper class for HuggingFace's RobertaTokenizerFast, that is intended for featurizing chemical reaction datasets. The featurizer computes the source and target required for a seq2seq task and applies the RobertaTokenizer on them separately. Additionally, it can also separate or mix the reactants and reagents before tokenizing.

Examples

```
>>> from deepchem.feat import RxnFeaturizer
>>> from transformers import RobertaTokenizerFast
>>> tokenizer = RobertaTokenizerFast.from_pretrained("seyonec/PubChem10M_SMILES_BPE_
↪450k")
>>> featurizer = RxnFeaturizer(tokenizer, sep_reagent=True)
>>> feats = featurizer.featurize(['CCS(=O)(=O)Cl.OCCBr>CCN(CC)CC.CCOCC>
↪CCS(=O)(=O)OCCBr'])
```

Notes

- The featurize method expects a List of reactions.
- **Use the sep_reagent toggle to enable/disable reagent separation.**
  - True - Separate the reactants and reagents
  - False - Mix the reactants and reagents

`__init__`(*tokenizer: RobertaTokenizerFast*, *sep_reagent: bool*, *max_length: int = 100*)

Initialize a ReactionFeaturizer object.

Parameters

- **`tokenizer`** (*`RobertaTokenizerFast`*) – HuggingFace Tokenizer to be used for featurization.
- **`sep_reagent`** (*`bool`*) – Toggle to separate or mix the reactants and reagents.
- **`max_length`** (*`int, default 100`*) – Maximum length of padding

`featurize`(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **`datapoints`** (*`Iterable[Any]`*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **`log_every_n`** (*`int, default 1000`*) – Logs featurization progress every *log_every_n* steps.

Returns

A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

BindingPocketFeaturizer

class BindingPocketFeaturizer

Featurizes binding pockets with information about chemical environments.

In many applications, it's desirable to look at binding pockets on macromolecules which may be good targets for potential ligands or other molecules to interact with. A *BindingPocketFeaturizer* expects to be given a macromolecule, and a list of pockets to featurize on that macromolecule. These pockets should be of the form produced by a *dc.dock.BindingPocketFinder*, that is as a list of *dc.utils.CoordinateBox* objects.

The base featurization in this class's featurization is currently very simple and counts the number of residues of each type present in the pocket. It's likely that you'll want to overwrite this implementation for more sophisticated downstream usecases. Note that this class's implementation will only work for proteins and not for other macromolecules

Note: This class requires mdtraj to be installed.

featurize(*protein_file: str*, *pockets: List[*CoordinateBox*]*) → ndarray

Calculate atomic coodinates.

Parameters

- **protein_file** (*str*) – Location of PDB file. Will be loaded by MDTraj
- **pockets** (*List[CoordinateBox]*) – List of *dc.utils.CoordinateBox* objects.

Returns

A numpy array of shale *(len(pockets), n_residues)*

Return type

np.ndarray

UserDefinedFeaturizer

class UserDefinedFeaturizer(*feature_fields*)

Directs usage of user-computed featurizations.

__init__(*feature_fields*)

Creates user-defined-featurizer.

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.
- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns

A numpy array containing a featurized representation of *datapoints*.

Return type

np.ndarray

DummyFeaturizer

class DummyFeaturizer

Class that implements a no-op featurization. This is useful when the raw dataset has to be used without featurizing the examples. The Molnet loader requires a featurizer input and such datasets can be used in their original form by passing the raw featurizer.

Examples

```
>>> import deepchem as dc
>>> smi_map = [["N#C[S-].O=C(CBr)c1ccc(C(F)(F)F)cc1>CCO.[K+]", "N
↪#CSCC(=O)c1ccc(C(F)(F)F)cc1"], ["C1COCCN1.FCC(Br)c1cccc(Br)n1>CCN(C(C)C)C(C)C.
↪CN(C)C=O.O", "FCC(c1cccc(Br)n1)N1CCOCC1"]]
>>> Featurizer = dc.feat.DummyFeaturizer()
>>> smi_feat = Featurizer.featurize(smi_map)
>>> smi_feat
array([['N#C[S-].O=C(CBr)c1ccc(C(F)(F)F)cc1>CCO.[K+]',
        'N#CSCC(=O)c1ccc(C(F)(F)F)cc1'],
       ['C1COCCN1.FCC(Br)c1cccc(Br)n1>CCN(C(C)C)C(C)C.CN(C)C=O.O',
        'FCC(c1cccc(Br)n1)N1CCOCC1']], dtype='<U55')
```

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Passes through dataset, and returns the datapoint.

Parameters

datapoints (*Iterable[Any]*) – A sequence of objects that you'd like to featurize.

Returns

datapoints – A numpy array containing a featurized representation of the datapoints.

Return type

np.ndarray

3.10.9 Base Featurizers (for develop)

Featurizer

The `dc.feat.Featurizer` class is the abstract parent class for all featurizers.

class Featurizer

Abstract class for calculating a set of features for a datapoint.

This class is abstract and cannot be invoked directly. You'll likely only interact with this class if you're a developer. In that case, you might want to make a child class which implements the *_featurize* method for calculating features for a single datapoints if you'd like to make a featurizer for a new datatype.

featurize(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for datapoints.

Parameters

- **datapoints** (*Iterable[Any]*) – A sequence of objects that you'd like to featurize. Subclassses of *Featurizer* should instantiate the *_featurize* method that featurizes objects in the sequence.

- **log_every_n** (*int, default 1000*) – Logs featurization progress every *log_every_n* steps.

Returns
A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

MolecularFeaturizer

If you're creating a new featurizer that featurizes molecules, you will want to inherit from the abstract `MolecularFeaturizer` base class. This featurizer can take RDKit mol objects or SMILES as inputs.

class MolecularFeaturizer(*use_original_atoms_order=False*)

Abstract class for calculating a set of features for a molecule.

The defining feature of a *MolecularFeaturizer* is that it uses SMILES strings and RDKit molecule objects to represent small molecules. All other featurizers which are subclasses of this class should plan to process input which comes as smiles strings or RDKit molecules.

Child classes need to implement the _featurize method for calculating features for a single molecule.

The subclasses of this class require RDKit to be installed.

__init__(*use_original_atoms_order=False*)

Parameters
use_original_atoms_order (*bool, default False*) – Whether to use original atom ordering or canonical ordering (default)

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → ndarray

Calculate features for molecules.

Parameters

- **datapoints** (*rdkit.Chem.rdchem.Mol / SMILES string / iterable*) – RDKit Mol, or SMILES string or iterable sequence of RDKit mols/SMILES strings.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns
features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

MaterialCompositionFeaturizer

If you're creating a new featurizer that featurizes compositional formulas, you will want to inherit from the abstract `MaterialCompositionFeaturizer` base class.

class MaterialCompositionFeaturizer

Abstract class for calculating a set of features for an inorganic crystal composition.

The defining feature of a *MaterialCompositionFeaturizer* is that it operates on 3D crystal chemical compositions. Inorganic crystal compositions are represented by Pymatgen composition objects. Featurizers for inorganic crystal compositions that are subclasses of this class should plan to process input which comes as Pymatgen composition objects.

This class is abstract and cannot be invoked directly. You'll likely only interact with this class if you're a developer. Child classes need to implement the _featurize method for calculating features for a single crystal composition.

Note: Some subclasses of this class will require pymatgen and matminer to be installed.

`featurize`(*datapoints: Iterable[str] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal compositions.

Parameters

- **datapoints** (*Iterable[str]*) – Iterable sequence of composition strings, e.g. "MoS2".
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *compositions*.

Return type

np.ndarray

MaterialStructureFeaturizer

If you're creating a new featurizer that featurizes inorganic crystal structure, you will want to inherit from the abstract `MaterialCompositionFeaturizer` base class. This featurizer can take pymatgen structure objects or dictionaries as inputs.

class MaterialStructureFeaturizer

Abstract class for calculating a set of features for an inorganic crystal structure.

The defining feature of a *MaterialStructureFeaturizer* is that it operates on 3D crystal structures with periodic boundary conditions. Inorganic crystal structures are represented by Pymatgen structure objects. Featurizers for inorganic crystal structures that are subclasses of this class should plan to process input which comes as pymatgen structure objects.

This class is abstract and cannot be invoked directly. You'll likely only interact with this class if you're a developer. Child classes need to implement the _featurize method for calculating features for a single crystal structure.

Note: Some subclasses of this class will require pymatgen and matminer to be installed.

`featurize`(*datapoints: Iterable[Dict[str, Any] | Any] | None = None*, *log_every_n: int = 1000*, ***kwargs*) → ndarray

Calculate features for crystal structures.

Parameters

- **datapoints** (*Iterable[Union[Dict, pymatgen.core.Structure]]*) – Iterable sequence of pymatgen structure dictionaries or pymatgen.core.Structure. Please confirm the dictionary representations of pymatgen.core.Structure from https://pymatgen.org/pymatgen.core.structure.html.
- **log_every_n** (*int, default 1000*) – Logging messages reported every *log_every_n* samples.

Returns

features – A numpy array containing a featurized representation of *datapoints*.

Return type
np.ndarray

ComplexFeaturizer

If you're creating a new featurizer that featurizes a pair of ligand molecules and proteins, you will want to inherit from the abstract `ComplexFeaturizer` base class. This featurizer can take a pair of PDB or SDF files which contain ligand molecules and proteins.

`class ComplexFeaturizer`

" Abstract class for calculating features for mol/protein complexes.

`featurize`(*datapoints: Iterable[Tuple[str, str]] | None = None*, *log_every_n: int = 100*, ***kwargs*) → ndarray

Calculate features for mol/protein complexes. :param datapoints: List of filenames (PDB, SDF, etc.) for ligand molecules and proteins.

Each element should be a tuple of the form (ligand_filename, protein_filename).

Returns
features – Array of features

Return type
np.ndarray

VocabularyBuilder

If you're creating a vocabulary builder for generating vocabulary from a corpus or input data, the vocabulary builder must inhere from `VocabularyBuilder` base class.

`class VocabularyBuilder`

Abstract class for building a vocabulary from a dataset.

`build`(*dataset:* Dataset)

Builds vocabulary from a dataset

Parameters
dataset (`Dataset`) – dataset to build vocabulary from.

`classmethod load`(*fname: str*)

Loads vocabulary from the specified file

Parameters
fname (`str`) – Path containing pre-build vocabulary.

`save`(*fname: str*)

Dump vocabulary to the specified file.

Parameters
fname (`str`) – A json file fname to save vocabulary.

`extend`(*dataset:* Dataset)

Extends vocabulary from a dataset

Parameters
dataset (`Dataset`) – dataset used for extending vocabulary

HuggingFaceVocabularyBuilder

A wrapper class for building vocabulary from algorithms implemented in tokenizers library.

hf_vocab

alias of <module 'deepchem.feat.vocabulary_builders.hf_vocab' from '/home/docs/checkouts/readthedocs.org/user_builds/deepche

3.11 Splitters

DeepChem `dc.splits.Splitter` objects are a tool to meaningfully split DeepChem datasets for machine learning testing. The core idea is that when evaluating a machine learning model, it's useful to creating training, validation and test splits of your source data. The training split is used to train models, the validation is used to benchmark different model architectures. The test is ideally held out till the very end when it's used to gauge a final estimate of the model's performance.

The `dc.splits` module contains a collection of scientifically aware splitters. In many cases, we want to evaluate scientific deep learning models more rigorously than standard deep models since we're looking for the ability to generalize to new domains. Some of the implemented splitters here may help.

Contents

- *General Splitters*
  - *RandomSplitter*
  - *RandomGroupSplitter*
  - *RandomStratifiedSplitter*
  - *SingletaskStratifiedSplitter*
  - *IndexSplitter*
  - *SpecifiedSplitter*
  - *TaskSplitter*
- *Molecule Splitters*
  - *ScaffoldSplitter*
  - *MolecularWeightSplitter*
  - *MaxMinSplitter*
  - *ButinaSplitter*
  - *FingerprintSplitter*
- *Base Splitter (for develop)*

3.11.1 General Splitters

RandomSplitter

class RandomSplitter

Class for doing random data splits.

Examples

```
>>> import numpy as np
>>> import deepchem as dc
>>> # Creating a dummy NumPy dataset
>>> X, y = np.random.randn(5), np.random.randn(5)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> # Creating a RandomSplitter object
>>> splitter = dc.splits.RandomSplitter()
>>> # Splitting dataset into train and test datasets
>>> train_dataset, test_dataset = splitter.train_test_split(dataset)
```

split(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

Splits internal compounds randomly into train/validation/test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns

A tuple of train indices, valid indices, and test indices. Each indices is a numpy array.

Return type

Tuple[np.ndarray, np.ndarray, np.ndarray]

__repr__() → str

Convert self to repr representation.

Returns

The string represents the class.

Return type

str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (`Dataset`) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.

- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

RandomGroupSplitter

class RandomGroupSplitter(*groups: Sequence*)

Random split based on groupings.

A splitter class that splits on groupings. An example use case is when there are multiple conformations of the same molecule that share the same topology. This splitter subsequently guarantees that resulting splits preserve groupings.

Note that it doesn't do any dynamic programming or something fancy to try to maximize the choice such that frac_train, frac_valid, or frac_test is maximized. It simply permutes the groups themselves. As such, use with caution if the number of elements per group varies significantly.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> X=np.arange(12)
>>> groups = [0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3]
>>> splitter = dc.splits.RandomGroupSplitter(groups=groups)
>>> dataset = dc.data.NumpyDataset(X)   # 12 elements
>>> train, test = splitter.train_test_split(dataset, frac_train=0.75, seed=0)
>>> print (train.ids) #array([6, 7, 8, 9, 10, 11, 3, 4, 5], dtype=object)
[6 7 8 9 10 11 3 4 5]
```

__init__(*groups: Sequence*)

Initialize this object.

Parameters

groups (*Sequence*) – An array indicating the group of each item. The length is equals to *len(dataset.X)*

Note: The examples of groups is the following.

groups : 3 2 2 0 1 1 2 4 3
dataset.X : 0 1 2 3 4 5 6 7 8

groups : a b b e q x a a r
dataset.X : 0 1 2 3 4 5 6 7 8

---

split(*dataset: Dataset, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int | None = None*) → Tuple[List[int], List[int], List[int]]

Return indices for specified split

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.

- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns
A tuple *(train_inds, valid_inds, test_inds* of the indices (integers) for the various splits.

Return type
Tuple[List[int], List[int], List[int]]

k_fold_split(*dataset: Dataset, k: int, directories: List[str] | None = None, **kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset: Dataset, train_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, seed: int | None = None, **kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns

A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type

Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

RandomStratifiedSplitter

class RandomStratifiedSplitter

RandomStratified Splitter class.

For sparse multitask datasets, a standard split offers no guarantees that the splits will have any active compounds. This class tries to arrange that each split has a proportional number of the actives for each task. This is strictly guaranteed only for single-task datasets, but for sparse multitask datasets it usually manages to produces a fairly accurate division of the actives for each task.

Note: This splitter is primarily designed for boolean labeled data. It considers only whether a label is zero or non-zero. When labels can take on multiple non-zero values, it does not try to give each split a proportional fraction of the samples with each value.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> from typing import Sequence
>>> # creation of demo data set with some smiles strings
>>> smiles= ['C', 'CC', 'CCC', 'CCCC', 'CCCCC']
>>> Xs = np.zeros(len(smiles))
>>> # creation of a deepchem dataset with the smile codes in the ids field
>>> dataset = dc.data.DiskDataset.from_numpy(X=Xs,ids=smiles)
>>> randomstratifiedsplitter = dc.splits.RandomStratifiedSplitter()
>>> train_dataset, test_dataset = randomstratifiedsplitter.train_test_split(dataset)
```

split(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple

Return indices for specified split

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to be split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **log_every_n** (*int, optional (default None)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns

A tuple *(train_inds, valid_inds, test_inds)* of the indices (integers) for the various splits.

Return type

Tuple

__repr__() → str

Convert self to repr representation.

Returns

The string represents the class.

Return type

str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset:* Dataset, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (`Dataset`) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset:* Dataset, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.

- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

SingletaskStratifiedSplitter

class SingletaskStratifiedSplitter(*task_number: int = 0*)

Class for doing data splits by stratification on a single task.

Examples

```
>>> n_samples = 100
>>> n_features = 10
>>> n_tasks = 10
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.rand(n_samples, n_tasks)
>>> w = np.ones_like(y)
>>> dataset = DiskDataset.from_numpy(np.ones((100,n_tasks)), np.ones((100,n_tasks)))
>>> splitter = SingletaskStratifiedSplitter(task_number=5)
>>> train_dataset, test_dataset = splitter.train_test_split(dataset)
```

__init__(*task_number: int = 0*)

Creates splitter object.

Parameters

task_number (*int, optional (default 0)*) – Task number for stratification.

k_fold_split(*dataset: Dataset, k: int, directories: List[str] | None = None, seed: int | None = None, log_every_n: int | None = None, **kwargs*) → List[*Dataset*]

Splits compounds into k-folds using stratified sampling. Overriding base class k_fold_split.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length k filepaths to save the result disk-datasets.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns

fold_datasets – List of dc.data.Dataset objects

Return type

List[*Dataset*]

split(*dataset: Dataset, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

Splits compounds into train/validation/test using stratified sampling.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – Fraction of dataset put into training data.

- **frac_valid** (*float, optional (default 0.1)*) – Fraction of dataset put into validation data.
- **frac_test** (*float, optional (default 0.1)*) – Fraction of dataset put into test data.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns
A tuple of train indices, valid indices, and test indices. Each indices is a numpy array.

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray]

train_test_split(*dataset: Dataset, train_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, seed: int | None = None, **kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset, train_dir: str | None = None, valid_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int = 1000, **kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*

- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

IndexSplitter

class IndexSplitter

Class for simple order based splits.

Use this class when the *Dataset* you have is already ordered sa you would like it to be processed. Then the first *frac_train* proportion is used for training, the next *frac_valid* for validation, and the final *frac_test* for testing. This class may make sense to use your *Dataset* is already time ordered (for example).

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> n_samples = 5
>>> n_features = 2
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.rand(n_samples)
>>> indexsplitter = dc.splits.IndexSplitter()
>>> dataset = dc.data.NumpyDataset(X, y)
>>> train_dataset, test_dataset = indexsplitter.train_test_split(dataset)
>>> print(train_dataset.ids)
[0 1 2 3]
>>> print (test_dataset.ids)
[4]
```

split(*dataset: Dataset, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

Splits internal compounds into train/validation/test in provided order.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional*) – Log every n examples (not currently used).

Returns
A tuple of train indices, valid indices, and test indices. Each indices is a numpy array.

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray]

__repr__() → str

Convert self to repr representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*

- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

SpecifiedSplitter

class SpecifiedSplitter(*valid_indices: List[int] | None = None*, *test_indices: List[int] | None = None*)

Split data in the fashion specified by user.

For some applications, you will already know how you'd like to split the dataset. In this splitter, you simplify specify *valid_indices* and *test_indices* and the datapoints at those indices are pulled out of the dataset. Note that this is different from *IndexSplitter* which only splits based on the existing dataset ordering, while this *SpecifiedSplitter* can split on any specified ordering.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.rand(n_samples, n_tasks)
>>> splitter = dc.splits.SpecifiedSplitter(valid_indices=[1,3,5], test_indices=[0,2,
↪7,9])
>>> dataset = dc.data.NumpyDataset(X, y)
>>> train_dataset, valid_dataset, test_dataset = splitter.train_valid_test_
↪split(dataset)
>>> print(train_dataset.ids)
[4 6 8]
>>> print(valid_dataset.ids)
[1 3 5]
```

(continues on next page)

(continued from previous page)

```
>>> print(test_dataset.ids)
[0 2 7 9]
```

`__init__`(*valid_indices: List[int] | None = None*, *test_indices: List[int] | None = None*)

Parameters

- **`valid_indices`** (*`List[int]`*) – List of indices of samples in the valid set
- **`test_indices`** (*`List[int]`*) – List of indices of samples in the test set

`split`(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

Splits internal compounds into train/validation/test in designated order.

Parameters

- **`dataset`** (`Dataset`) – Dataset to be split.
- **`frac_train`** (*`float, optional (default 0.8)`*) – Fraction of dataset put into training data.
- **`frac_valid`** (*`float, optional (default 0.1)`*) – Fraction of dataset put into validation data.
- **`frac_test`** (*`float, optional (default 0.1)`*) – Fraction of dataset put into test data.
- **`seed`** (*`int, optional (default None)`*) – Random seed to use.
- **`log_every_n`** (*`int, optional (default None)`*) – Log every n examples (not currently used).

Returns

A tuple of train indices, valid indices, and test indices. Each indices is a numpy array.

Return type

Tuple[np.ndarray, np.ndarray, np.ndarray]

`k_fold_split`(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **`dataset`** (`Dataset`) – Dataset to do a k-fold split
- **`k`** (*`int`*) – Number of folds to split *dataset* into.
- **`directories`** (*`List[str], optional (default None)`*) – List of length 2*k filepaths to save the result disk-datasets.

Returns

List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type

List[Tuple[*Dataset*, *Dataset*]]

`train_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset:* Dataset, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

TaskSplitter

class TaskSplitter

Provides a simple interface for splitting datasets task-wise.

For some learning problems, the training and test datasets should have different tasks entirely. This is a different paradigm from the usual Splitter, which ensures that split datasets have different datapoints, not different tasks.

`__init__`()

Creates Task Splitter object.

`train_valid_test_split`(*dataset*, *frac_train=0.8*, *frac_valid=0.1*, *frac_test=0.1*)

Performs a train/valid/test split of the tasks for dataset.

If split is uneven, spillover goes to test.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to be split
- **frac_train** (*float, optional*) – Proportion of tasks to be put into train. Rounded to nearest int.
- **frac_valid** (*float, optional*) – Proportion of tasks to be put into valid. Rounded to nearest int.
- **frac_test** (*float, optional*) – Proportion of tasks to be put into test. Rounded to nearest int.

`k_fold_split`(*dataset*, *K*)

Performs a K-fold split of the tasks for dataset.

If split is uneven, spillover goes to last fold.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to be split
- **K** (*int*) – Number of splits to be made

`split`(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple

Return indices for specified split

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to be split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.

- **log_every_n** (*int, optional (default None)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple *(train_inds, valid_inds, test_inds)* of the indices (integers) for the various splits.

Return type
Tuple

train_test_split(*dataset: Dataset, train_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, seed: int | None = None, **kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

3.11.2 Molecule Splitters

ScaffoldSplitter

class ScaffoldSplitter

Class for doing data splits based on the scaffold of small molecules.

Group molecules based on the Bemis-Murcko scaffold representation, which identifies rings, linkers, frameworks (combinations between linkers and rings) and atomic properties such as atom type, hibridization and bond order in a dataset of molecules. Then split the groups by the number of molecules in each group in decreasing order.

It is necessary to add the smiles representation in the ids field during the DiskDataset creation.

Examples

```
>>> import deepchem as dc
>>> # creation of demo data set with some smiles strings
... data_test= ["CC(C)Cl" , "CCC(C)CO" ,  "CCCCCCCO" , "CCCCCCCC(=O)OC" ,
↪"c3ccc2nc1ccccc1cc2c3" , "Nc2cccc3nc1ccccc1cc23" , "C1CCCCCC1" ]
>>> Xs = np.zeros(len(data_test))
>>> Ys = np.ones(len(data_test))
>>> # creation of a deepchem dataset with the smile codes in the ids field
... dataset = dc.data.DiskDataset.from_numpy(X=Xs,y=Ys,w=np.zeros(len(data_test)),
↪ids=data_test)
>>> scaffoldsplitter = dc.splits.ScaffoldSplitter()
>>> train,test = scaffoldsplitter.train_test_split(dataset)
>>> train
<DiskDataset X.shape: (5,), y.shape: (5,), w.shape: (5,), ids: ['CC(C)Cl' 'CCC(C)CO
↪' 'CCCCCCCO' 'CCCCCCCC(=O)OC' 'C1CCCCCC1'], task_names: [0]>
```

References

Notes

- This class requires RDKit to be installed.
- When a SMILES representation of a molecule is invalid, the splitter skips processing

the datapoint i.e it will not include the molecule in any splits.

split(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = 1000*) → Tuple[List[int], List[int], List[int]]

Splits internal compounds into train/validation/test by scaffold.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns

A tuple of train indices, valid indices, and test indices. Each indices is a list of integers.

Return type

Tuple[List[int], List[int], List[int]]

generate_scaffolds(*dataset: Dataset*, *log_every_n: int = 1000*) → List[List[int]]

Returns all scaffolds from the dataset.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
scaffold_sets – List of indices of each scaffold in the dataset.

Return type
List[List[int]]

__repr__() → str

Convert self to repr representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset: Dataset, k: int, directories: List[str] | None = None, **kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

`train_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

`train_valid_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.

- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

MolecularWeightSplitter

class MolecularWeightSplitter

Class for doing data splits by molecular weight.

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> # creation of demo data set with some smiles strings
>>> smiles= ['C', 'CC', 'CCC', 'CCCC', 'CCCCC']
>>> Xs = np.zeros(len(smiles))
>>> # creation of a deepchem dataset with the smile codes in the ids field
>>> dataset = dc.data.DiskDataset.from_numpy(X=Xs,ids=smiles)
>>> molecularweightsplitter = dc.splits.MolecularWeightSplitter()
>>> train_dataset, test_dataset = molecularweightsplitter.train_test_split(dataset)
>>> print(train_dataset.ids)
['C' 'CC' 'CCC' 'CCCC']
>>> print(test_dataset.ids)
['CCCCC']
```

split(*dataset: Dataset, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

Splits on molecular weight.

Splits internal compounds into train/validation/test using the MW calculated by SMILES string.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.

- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns
A tuple of train indices, valid indices, and test indices. Each indices is a numpy array.

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray]

__repr__() → str

Convert self to repr representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (`data like object`) – Dataset to be split.
- **train_dir** (`str, optional (default None)`) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (`str, optional (default None)`) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (`float, optional (default 0.8)`) – The fraction of data to be used for the training split.
- **seed** (`int, optional (default None)`) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (`str, optional (default None)`) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (`str, optional (default None)`) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (`str, optional (default None)`) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (`float, optional (default 0.8)`) – The fraction of data to be used for the training split.

- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

MaxMinSplitter

class MaxMinSplitter

Chemical diversity splitter.

Class for doing splits based on the MaxMin diversity algorithm. Intuitively, the test set is comprised of the most diverse compounds of the entire dataset. Furthermore, the validation set is comprised of diverse compounds under the test set.

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> # creation of demo data set with some smiles strings
>>> smiles= ['C', 'CC', 'CCC', 'CCCC', 'CCCCC']
>>> Xs = np.zeros(len(smiles))
>>> # creation of a deepchem dataset with the smile codes in the ids field
>>> dataset = dc.data.DiskDataset.from_numpy(X=Xs,ids=smiles)
>>> maxminsplitter = dc.splits.MaxMinSplitter()
>>> train_dataset, test_dataset = maxminsplitter.train_test_split(dataset)
```

split(*dataset:* Dataset, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int* | *None = None*, *log_every_n: int* | *None = None*) → Tuple[List[int], List[int], List[int]]

Splits internal compounds into train/validation/test using the MaxMin diversity algorithm.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.

- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns
A tuple of train indices, valid indices, and test indices. Each indices is a list of integers.

Return type
Tuple[List[int], List[int], List[int]]

__repr__() → str

Convert self to repr representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

__str__() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

__weakref__

list of weak references to the object (if defined)

k_fold_split(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

`train_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (`data like object`) – Dataset to be split.
- **train_dir** (`str, optional (default None)`) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (`str, optional (default None)`) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (`float, optional (default 0.8)`) – The fraction of data to be used for the training split.
- **seed** (`int, optional (default None)`) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

`train_valid_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (`str, optional (default None)`) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (`str, optional (default None)`) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (`str, optional (default None)`) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (`float, optional (default 0.8)`) – The fraction of data to be used for the training split.
- **frac_valid** (`float, optional (default 0.1)`) – The fraction of data to be used for the validation split.

- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

ButinaSplitter

class ButinaSplitter(*cutoff: float = 0.6*)

Class for doing data splits based on the butina clustering of a bulk tanimoto fingerprint matrix.

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> # creation of demo data set with some smiles strings
>>> smiles= ['C', 'CC', 'CCC', 'CCCC', 'CCCCC']
>>> Xs = np.zeros(len(smiles))
>>> # creation of a deepchem dataset with the smile codes in the ids field
>>> dataset = dc.data.DiskDataset.from_numpy(X=Xs,ids=smiles)
>>> butinasplitter = dc.splits.ButinaSplitter()
>>> train_dataset, test_dataset = butinasplitter.train_test_split(dataset)
>>> print(train_dataset.ids)
['CCCC' 'CCC' 'CCCCC' 'CC']
>>> print(test_dataset.ids)
['C']
```

__init__(*cutoff: float = 0.6*)

Create a ButinaSplitter.

Parameters
cutoff (*float (default 0.6)*) – The cutoff value for tanimoto similarity. Molecules that are more similar than this will tend to be put in the same dataset.

split(*dataset:* Dataset, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple[List[int], List[int], List[int]]

Splits internal compounds into train and validation based on the butina clustering algorithm. This splitting algorithm has an O(N^2) run time, where N is the number of elements in the dataset. The dataset is expected to be a classification dataset.

This algorithm is designed to generate validation data that are novel chemotypes. Setting a small cutoff value will generate smaller, finer clusters of high similarity, whereas setting a large cutoff value will generate larger, coarser clusters of low similarity.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns
A tuple of train indices, valid indices, and test indices.

Return type
Tuple[List[int], List[int], List[int]]

k_fold_split(*dataset: Dataset, k: int, directories: List[str] | None = None, **kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (Dataset) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_test_split(*dataset: Dataset, train_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, seed: int | None = None, **kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

`train_valid_test_split`(*dataset:* Dataset, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

FingerprintSplitter

`class FingerprintSplitter`

Class for doing data splits based on the Tanimoto similarity between ECFP4 fingerprints.

This class tries to split the data such that the molecules in each dataset are as different as possible from the ones in the other datasets. This makes it a very stringent test of models. Predicting the test and validation sets may require extrapolating far outside the training data.

The running time for this splitter scales as O(n^2) in the number of samples. Splitting large datasets can take a long time.

Note: This class requires RDKit to be installed.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> # creation of demo data set with some smiles strings
>>> smiles= ['C', 'CC', 'CCC', 'CCCC', 'CCCCC']
>>> Xs = np.zeros(len(smiles))
>>> # creation of a deepchem dataset with the smile codes in the ids field
>>> dataset = dc.data.DiskDataset.from_numpy(X=Xs,ids=smiles)
>>> fingerprintsplitter = dc.splits.FingerprintSplitter()
>>> train_dataset, test_dataset = fingerprintsplitter.train_test_split(dataset)
>>> print(train_dataset.ids)
['C' 'CCCCC' 'CCCC' 'CCC']
>>> print(test_dataset.ids)
['CC']
```

split(*dataset: Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int | None = None*) → Tuple[List[int], List[int], List[int]]

Splits compounds into training, validation, and test sets based on the Tanimoto similarity of their ECFP4 fingerprints. This splitting algorithm has an O(N^2) run time, where N is the number of elements in the dataset.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use (ignored since this algorithm is deterministic).
- **log_every_n** (*int, optional (default None)*) – Log every n examples (not currently used).

Returns

A tuple of train indices, valid indices, and test indices.

Return type

Tuple[List[int], List[int], List[int]]

__repr__() → str

Convert self to repr representation.

Returns

The string represents the class.

Return type

str

Examples

```
>>> import deepchem as dc
>>> dc.splits.RandomSplitter()
RandomSplitter[]
```

`__str__`() → str

Convert self to str representation.

Returns
The string represents the class.

Return type
str

Examples

```
>>> import deepchem as dc
>>> str(dc.splits.RandomSplitter())
'RandomSplitter'
```

`__weakref__`

list of weak references to the object (if defined)

`k_fold_split`(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **`dataset`** (`Dataset`) – Dataset to do a k-fold split
- **`k`** (*`int`*) – Number of folds to split *dataset* into.
- **`directories`** (*`List[str], optional (default None)`*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

`train_test_split`(*dataset: Dataset*, *train_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *seed: int | None = None*, ***kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **`dataset`** (*`data like object`*) – Dataset to be split.
- **`train_dir`** (*`str, optional (default None)`*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **`test_dir`** (*`str, optional (default None)`*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.

- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

3.11.3 Base Splitter (for develop)

The `dc.splits.Splitter` class is the abstract parent class for all splitters. This class should never be directly instantiated.

class Splitter

Splitters split up Datasets into pieces for training/validation/testing.

In machine learning applications, it's often necessary to split up a dataset into training/validation/test sets. Or to k-fold split a dataset (that is, divide into k equal subsets) for cross-validation. The *Splitter* class is an abstract superclass for all splitters that captures the common API across splitter classes.

Note that *Splitter* is an abstract superclass. You won't want to instantiate this class directly. Rather you will want to use a concrete subclass for your application.

k_fold_split(*dataset: Dataset*, *k: int*, *directories: List[str] | None = None*, ***kwargs*) → List[Tuple[*Dataset*, *Dataset*]]

Parameters

- **dataset** (`Dataset`) – Dataset to do a k-fold split
- **k** (*int*) – Number of folds to split *dataset* into.
- **directories** (*List[str], optional (default None)*) – List of length 2*k filepaths to save the result disk-datasets.

Returns
List of length k tuples of (train, cv) where *train* and *cv* are both *Dataset*.

Return type
List[Tuple[*Dataset*, *Dataset*]]

train_valid_test_split(*dataset: Dataset*, *train_dir: str | None = None*, *valid_dir: str | None = None*, *test_dir: str | None = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: int | None = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[*Dataset*, *Dataset*, *Dataset*]

Splits self into train/validation/test sets.

Returns Dataset objects for train, valid, test.

Parameters

- **dataset** (`Dataset`) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)*
- **valid_dir** (*str, optional (default None)*) – If specified, the directory in which the generated valid dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.

- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **log_every_n** (*int, optional (default 1000)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple of train, valid and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, Optional[*Dataset*], *Dataset*]

train_test_split(*dataset: Dataset, train_dir: str | None = None, test_dir: str | None = None, frac_train: float = 0.8, seed: int | None = None, **kwargs*) → Tuple[*Dataset*, *Dataset*]

Splits self into train/test sets.

Returns Dataset objects for train/test.

Parameters

- **dataset** (*data like object*) – Dataset to be split.
- **train_dir** (*str, optional (default None)*) – If specified, the directory in which the generated training dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **test_dir** (*str, optional (default None)*) – If specified, the directory in which the generated test dataset should be stored. This is only considered if *isinstance(dataset, dc.data.DiskDataset)* is True.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **seed** (*int, optional (default None)*) – Random seed to use.

Returns
A tuple of train and test datasets as dc.data.Dataset objects.

Return type
Tuple[*Dataset*, *Dataset*]

split(*dataset: Dataset, frac_train: float = 0.8, frac_valid: float = 0.1, frac_test: float = 0.1, seed: int | None = None, log_every_n: int | None = None*) → Tuple

Return indices for specified split

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to be split.
- **seed** (*int, optional (default None)*) – Random seed to use.
- **frac_train** (*float, optional (default 0.8)*) – The fraction of data to be used for the training split.
- **frac_valid** (*float, optional (default 0.1)*) – The fraction of data to be used for the validation split.
- **frac_test** (*float, optional (default 0.1)*) – The fraction of data to be used for the test split.
- **log_every_n** (*int, optional (default None)*) – Controls the logger by dictating how often logger outputs will be produced.

Returns
A tuple *(train_inds, valid_inds, test_inds)* of the indices (integers) for the various splits.

Return type
Tuple

3.12 Transformers

DeepChem `dc.trans.Transformer` objects are another core building block of DeepChem programs. Often times, machine learning systems are very delicate. They need their inputs and outputs to fit within a pre-specified range or follow a clean mathematical distribution. Real data of course is wild and hard to control. What do you do if you have a crazy dataset and need to bring its statistics to heel? Fear not for you have `Transformer` objects.

Contents

- *General Transformers*
  - *NormalizationTransformer*
  - *MinMaxTransformer*
  - *ClippingTransformer*
  - *LogTransformer*
  - *CDFTransformer*
  - *PowerTransformer*
  - *BalancingTransformer*
  - *DuplicateBalancingTransformer*
  - *ImageTransformer*
  - *FeaturizationTransformer*
- *Specified Usecase Transformers*
  - *CoulombFitTransformer*
  - *IRVTransformer*
  - *DAGTransformer*
  - *RxnSplitTransformer*
- *Base Transformer (for develop)*

3.12.1 General Transformers

NormalizationTransformer

class NormalizationTransformer(*transform_X: bool = False*, *transform_y: bool = False*, *transform_w: bool = False*, *dataset: Dataset | None = None*, *transform_gradients: bool = False*, *move_mean: bool = True*)

Normalizes dataset to have zero mean and unit standard deviation

This transformer transforms datasets to have zero mean and unit standard deviation.

Examples

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.rand(n_samples, n_tasks)
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.NormalizationTransformer(transform_y=True,␣
↪dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

Note: This class can only transform *X* or *y* and not *w*. So only one of *transform_X* or *transform_y* can be set.

Raises

ValueError – if *transform_X* and *transform_y* are both set.

__init__(*transform_X: bool = False*, *transform_y: bool = False*, *transform_w: bool = False*, *dataset: Dataset | None = None*, *transform_gradients: bool = False*, *move_mean: bool = True*)

Initialize normalization transformation.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **transform_w** (*bool, optional (default False)*) – Whether to transform w
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed

transform_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels

- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of ids.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*z: ndarray*) → ndarray

Undo transformation on provided data.

Parameters
z (*np.ndarray*) – Array to transform back

Returns
z_out – Array with normalization undone.

Return type
np.ndarray

untransform_grad(*grad*, *tasks*)

DEPRECATED. DO NOT USE.

transform(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.

- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

MinMaxTransformer

class **MinMaxTransformer**(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*)

Ensure each value rests between 0 and 1 by using the min and max.

MinMaxTransformer transforms the dataset by shifting each axis of X or y (depending on whether transform_X or transform_y is True), except the first one by the minimum value along the axis and dividing the result by the range (maximum value - minimum value) along the axis. This ensures each axis is between 0 and 1. In case of multi-task learning, it ensures each task is given equal importance.

Given original array A, the transformed array can be written as:

```
>>> import numpy as np
>>> A = np.random.rand(10, 10)
>>> A_min = np.min(A, axis=0)
>>> A_max = np.max(A, axis=0)
>>> A_t = np.nan_to_num((A - A_min)/(A_max - A_min))
```

Examples

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.rand(n_samples, n_tasks)
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.MinMaxTransformer(transform_y=True, dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

Note: This class can only transform *X* or *y* and not *w*. So only one of *transform_X* or *transform_y* can be set.

Raises

ValueError – if *transform_X* and *transform_y* are both set.

__init__(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*)

Initialization of MinMax transformer.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed

transform_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of ids.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*z: ndarray*) → ndarray

Undo transformation on provided data.

Parameters
z (*np.ndarray*) – Transformed X or y array

Returns
Array with min-max scaling undone.

Return type
np.ndarray

transform(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

ClippingTransformer

class ClippingTransformer(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *x_max: float = 5.0*, *y_max: float = 500.0*)

Clip large values in datasets.

Examples

Let's clip values from a synthetic dataset

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.zeros((n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.ClippingTransformer(transform_X=True)
>>> dataset = transformer.transform(dataset)
```

__init__(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *x_max: float = 5.0*, *y_max: float = 500.0*)

Initialize clipping transformation.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **dataset** (*dc.data.Dataset object, optional*) – Dataset to be transformed
- **x_max** (*float, optional*) – Maximum absolute value for X

- **y_max** (*float, optional*) – Maximum absolute value for y

---

Note: This transformer can transform *X* and *y* jointly, but does not transform *w*.

Raises
`ValueError` – if *transform_w* is set.

transform_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w) arrays.

Parameters

- **X** (*np.ndarray*) – Array of Features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights
- **ids** (*np.ndarray*) – Array of ids.

Returns

- **X** (*np.ndarray*) – Transformed features
- **y** (*np.ndarray*) – Transformed tasks
- **w** (*np.ndarray*) – Transformed weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*z: ndarray*) → ndarray

Not implemented.

transform(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

LogTransformer

`class LogTransformer`(*transform_X: bool = False*, *transform_y: bool = False*, *features: List[int] | None = None*, *tasks: List[str] | None = None*, *dataset: Dataset | None = None*)

Computes a logarithmic transformation

This transformer computes the transformation given by

```
>>> import numpy as np
>>> A = np.random.rand(10, 10)
>>> A = np.log(A + 1)
```

Assuming that tasks/features are not specified. If specified, then transformations are only performed on specified tasks/features.

Examples

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.zeros((n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.LogTransformer(transform_X=True)
>>> dataset = transformer.transform(dataset)
```

Note: This class can only transform *X* or *y* and not *w*. So only one of *transform_X* or *transform_y* can be set.

Raises

`ValueError` – if *transform_w* is set or *transform_X* and *transform_y* are both set.

`__init__`(*transform_X: bool = False*, *transform_y: bool = False*, *features: List[int] | None = None*, *tasks: List[str] | None = None*, *dataset:* Dataset *| None = None*)

Initialize log transformer.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **features** (*list[Int]*) – List of features indices to transform
- **tasks** (*list[str]*) – List of task names to transform.
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of weights.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*z: ndarray*) → ndarray

Undo transformation on provided data.

Parameters

z (*np.ndarray,*) – Transformed X or y array

Returns

Array with a logarithmic transformation undone.

Return type

np.ndarray

`transform`(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.

- **out_dir** (`str, optional`) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

CDFTransformer

class CDFTransformer(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *bins: int = 2*)

Histograms the data and assigns values based on sorted list.

Acts like a Cumulative Distribution Function (CDF). If given a dataset of samples from a continuous distribution computes the CDF of this dataset and replaces values with their corresponding CDF values.

Examples

Let's look at an example where we transform only features.

```
>>> N = 10
>>> n_feat = 5
>>> n_bins = 100
```

Note that we're using 100 bins for our CDF histogram

```
>>> import numpy as np
>>> X = np.random.normal(size=(N, n_feat))
>>> y = np.random.randint(2, size=(N,))
>>> dataset = dc.data.NumpyDataset(X, y)
>>> cdftrans = dc.trans.CDFTransformer(transform_X=True, dataset=dataset, bins=n_
```

(continues on next page)

(continued from previous page)

```
↪bins)
>>> dataset = cdftrans.transform(dataset)
```

Note that you can apply this transformation to *y* as well

```
>>> X = np.random.normal(size=(N, n_feat))
>>> y = np.random.normal(size=(N,))
>>> dataset = dc.data.NumpyDataset(X, y)
>>> cdftrans = dc.trans.CDFTransformer(transform_y=True, dataset=dataset, bins=n_
↪bins)
>>> dataset = cdftrans.transform(dataset)
```

`__init__`(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *bins: int = 2*)

Initialize this transformer.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed
- **bins** (*int, optional (default 2)*) – Number of bins to use when computing histogram.

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Performs CDF transform on data.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*z: ndarray*) → ndarray

Undo transformation on provided data.

Note that this transformation is only undone for y.

Parameters

z (*np.ndarray,*) – Transformed y array

Returns

Array with the transformation undone.

Return type
np.ndarray

`transform`(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

`transform_on_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

PowerTransformer

class `PowerTransformer`(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *powers: List[int] = [1]*)

Takes power n transforms of the data based on an input vector.

Computes the specified powers of the dataset. This can be useful if you're looking to add higher order features of the form *x_i^2*, *x_i^3* etc. to your dataset.

Examples

Let's look at an example where we transform only *X*.

```
>>> N = 10
>>> n_feat = 5
>>> powers = [1, 2, 0.5]
```

So in this example, we're taking the identity, squares, and square roots. Now let's construct our matrices

```
>>> import numpy as np
>>> X = np.random.rand(N, n_feat)
>>> y = np.random.normal(size=(N,))
>>> dataset = dc.data.NumpyDataset(X, y)
>>> trans = dc.trans.PowerTransformer(transform_X=True, dataset=dataset,␣
↪powers=powers)
>>> dataset = trans.transform(dataset)
```

Let's now look at an example where we transform *y*. Note that the *y* transform expands out the feature dimensions of *y* the same way it does for *X* so this transform is only well defined for singletask datasets.

```
>>> import numpy as np
>>> X = np.random.rand(N, n_feat)
>>> y = np.random.rand(N)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> trans = dc.trans.PowerTransformer(transform_y=True, dataset=dataset,␣
↪powers=powers)
>>> dataset = trans.transform(dataset)
```

__init__(*transform_X: bool = False*, *transform_y: bool = False*, *dataset: Dataset | None = None*, *powers: List[int] = [1]*)

Initialize this transformer

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed. Note that this argument is ignored since
- **specified.** (*PowerTransformer doesn't require it to be*) – powers: list[int], optional (default *[1]*) The list of powers of features/labels to compute.

transform_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Performs power transform on data.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*z: ndarray*) → ndarray

Undo transformation on provided data.

Parameters

`z` (`np.ndarray,`) – Transformed y array

Returns

Array with the power transformation undone.

Return type

np.ndarray

transform(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- `dataset` (`dc.data.Dataset`) – Dataset object to be transformed.
- `parallel` (`bool, optional (default False)`) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- `out_dir` (`str, optional`) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns

A newly transformed Dataset object

Return type

Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- `X` (`np.ndarray`) – Array of features
- `y` (`np.ndarray`) – Array of labels
- `w` (`np.ndarray`) – Array of weights.
- `ids` (`np.ndarray`) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights

- **idstrans** (*np.ndarray*) – Transformed array of ids

BalancingTransformer

class BalancingTransformer(*dataset: Dataset*)

Balance positive and negative (or multiclass) example weights.

This class balances the sample weights so that the sum of all example weights from all classes is the same. This can be useful when you're working on an imbalanced dataset where there are far fewer examples of some classes than others.

Examples

Here's an example for a binary dataset.

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> n_classes = 2
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.randint(n_classes, size=(n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.BalancingTransformer(dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

And here's a multiclass dataset example.

```
>>> n_samples = 50
>>> n_features = 3
>>> n_tasks = 1
>>> n_classes = 5
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.randint(n_classes, size=(n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.BalancingTransformer(dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

See also:

deepchem.trans.DuplicateBalancingTransformer
Balance by duplicating samples.

Note: This transformer is only meaningful for classification datasets where *y* takes on a limited set of values. This class can only transform *w* and does not transform *X* or *y*.

Raises

`ValueError` – if *transform_X* or *transform_y* are set. Also raises or if *y* or *w* aren't of shape *(N,)* or *(N, n_tasks)*.

`__init__`(*dataset:* Dataset)

Initializes transformation based on dataset statistics.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **transform_w** (*bool, optional (default False)*) – Whether to transform w
- **transform_ids** (*bool, optional (default False)*) – Whether to transform ids
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of weights.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform`(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns

A newly transformed Dataset object

Return type

Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*transformed: ndarray*) → ndarray

Reverses stored transformation on provided data.

Depending on whether *transform_X* or *transform_y* or *transform_w* was set, this will perform different un-transformations. Note that this method may not always be defined since some transformations aren't 1-1.

Parameters

transformed (*np.ndarray*) – Array which was previously transformed by this class.

DuplicateBalancingTransformer

class DuplicateBalancingTransformer(*dataset:* Dataset)

Balance binary or multiclass datasets by duplicating rarer class samples.

This class balances a dataset by duplicating samples of the rarer class so that the sum of all example weights from all classes is the same. (Up to integer rounding of course). This can be useful when you're working on an imabalanced dataset where there are far fewer examples of some classes than others.

This class differs from *BalancingTransformer* in that it actually duplicates rarer class samples rather than just increasing their sample weights. This may be more friendly for models that are numerically fragile and can't handle imbalanced example weights.

Examples

Here's an example for a binary dataset.

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> n_classes = 2
>>> import deepchem as dc
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.randint(n_classes, size=(n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.DuplicateBalancingTransformer(dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

And here's a multiclass dataset example.

```
>>> n_samples = 50
>>> n_features = 3
>>> n_tasks = 1
>>> n_classes = 5
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features)
>>> y = np.random.randint(n_classes, size=(n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> transformer = dc.trans.DuplicateBalancingTransformer(dataset=dataset)
>>> dataset = transformer.transform(dataset)
```

See also:

`deepchem.trans.BalancingTransformer`
Balance by changing sample weights.

Note: This transformer is only well-defined for singletask datasets. (Since examples are actually duplicated, there's no meaningful way to duplicate across multiple tasks in a way that preserves the balance.)

This transformer is only meaningful for classification datasets where *y* takes on a limited set of values. This class transforms all of *X*, *y*, *w*, *ids*.

Raises
`ValueError` –

`__init__`(*dataset:* Dataset)
Initializes transformation based on dataset statistics.

Parameters

- **`transform_X`** (*`bool, optional (default False)`*) – Whether to transform X
- **`transform_y`** (*`bool, optional (default False)`*) – Whether to transform y
- **`transform_w`** (*`bool, optional (default False)`*) – Whether to transform w
- **`transform_ids`** (*`bool, optional (default False)`*) – Whether to transform ids
- **`dataset`** (*`dc.data.Dataset object, optional (default None)`*) – Dataset to be transformed

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, id) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idtrans** (*np.ndarray*) – Transformed array of identifiers

transform(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns

A newly transformed Dataset object

Return type

Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels

- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*transformed: ndarray*) → ndarray

Reverses stored transformation on provided data.

Depending on whether *transform_X* or *transform_y* or *transform_w* was set, this will perform different un-transformations. Note that this method may not always be defined since some transformations aren't 1-1.

Parameters
`transformed` (*np.ndarray*) – Array which was previously transformed by this class.

ImageTransformer

class `ImageTransformer`(*size: Tuple[int, int]*)

Convert an image into width, height, channel

Note: This class require Pillow to be installed.

`__init__`(*size: Tuple[int, int]*)

Initializes ImageTransformer.

Parameters
`size` (*Tuple[int, int]*) – The image size, a tuple of (width, height).

`transform_array`(*X*, *y*, *w*)

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- `X` (*np.ndarray*) – Array of features
- `y` (*np.ndarray*) – Array of labels
- `w` (*np.ndarray*) – Array of weights.
- `ids` (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform`(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- `dataset` (*dc.data.Dataset*) – Dataset object to be transformed.

- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*transformed: ndarray*) → ndarray

Reverses stored transformation on provided data.

Depending on whether *transform_X* or *transform_y* or *transform_w* was set, this will perform different un-transformations. Note that this method may not always be defined since some transformations aren't 1-1.

Parameters
transformed (*np.ndarray*) – Array which was previously transformed by this class.

FeaturizationTransformer

class FeaturizationTransformer(*dataset:* Dataset | *None* = *None*, *featurizer:* Featurizer | *None* = *None*)

A transformer which runs a featurizer over the X values of a dataset.

Datasets used by this transformer must be compatible with the internal featurizer. The idea of this transformer is that it allows for the application of a featurizer to an existing dataset.

Examples

```
>>> smiles = ["C", "CC"]
>>> X = np.array(smiles)
>>> y = np.array([1, 0])
>>> dataset = dc.data.NumpyDataset(X, y)
>>> trans = dc.trans.FeaturizationTransformer(dataset, dc.feat.
↪CircularFingerprint())
>>> dataset = trans.transform(dataset)
```

`__init__`(*dataset: Dataset | None = None*, *featurizer: Featurizer | None = None*)

Initialization of FeaturizationTransformer

Parameters

- **`dataset`** (*`dc.data.Dataset object, optional (default None)`*) – Dataset to be transformed
- **`featurizer`** (*`dc.feat.Featurizer object, optional (default None)`*) – Featurizer applied to perform transformations.

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms arrays of rdkit mols using internal featurizer.

Parameters

- **`X`** (*`np.ndarray`*) – Array of features
- **`y`** (*`np.ndarray`*) – Array of labels
- **`w`** (*`np.ndarray`*) – Array of weights.
- **`ids`** (*`np.ndarray`*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform`(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **`dataset`** (*`dc.data.Dataset`*) – Dataset object to be transformed.
- **`parallel`** (*`bool, optional (default False)`*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **`out_dir`** (*`str, optional`*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*transformed: ndarray*) → ndarray

Reverses stored transformation on provided data.

Depending on whether *transform_X* or *transform_y* or *transform_w* was set, this will perform different un-transformations. Note that this method may not always be defined since some transformations aren't 1-1.

Parameters
transformed (*np.ndarray*) – Array which was previously transformed by this class.

3.12.2 Specified Usecase Transformers

CoulombFitTransformer

class CoulombFitTransformer(*dataset:* Dataset)

Performs randomization and binarization operations on batches of Coulomb Matrix features during fit.

Examples

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features, n_features)
>>> y = np.zeros((n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
```

(continues on next page)

(continued from previous page)

```
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> fit_transformers = [dc.trans.CoulombFitTransformer(dataset)]
>>> model = dc.models.MultitaskFitTransformRegressor(n_tasks,
...     [n_features, n_features], batch_size=n_samples, fit_transformers=fit_
↪transformers, n_evals=1)
>>> print(model.n_features)
12
```

`__init__`(*dataset: Dataset*)

Initializes CoulombFitTransformer.

Parameters
`dataset` (*`dc.data.Dataset`*) – Dataset object to be transformed.

`realize`(*X: ndarray*) → ndarray

Randomize features.

Parameters
`X` (*`np.ndarray`*) – Features

Returns
X – Randomized features

Return type
np.ndarray

`normalize`(*X: ndarray*) → ndarray

Normalize features.

Parameters
`X` (*`np.ndarray`*) – Features

Returns
X – Normalized features

Return type
np.ndarray

`expand`(*X: ndarray*) → ndarray

Binarize features.

Parameters
`X` (*`np.ndarray`*) – Features

Returns
X – Binarized features

Return type
np.ndarray

`X_transform`(*X: ndarray*) → ndarray

Perform Coulomb Fit transform on features.

Parameters
`X` (*`np.ndarray`*) – Features

Returns
X – Transformed features

Return type
np.ndarray

transform_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Performs randomization and binarization operations on data.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

untransform(*z: ndarray*) → ndarray

Not implemented.

transform(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.

- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

IRVTransformer

class IRVTransformer(*K: int*, *n_tasks: int*, *dataset:* Dataset)

Performs transform from ECFP to IRV features(K nearest neighbors).

This transformer is required by *MultitaskIRVClassifier* as a preprocessing step before training.

Examples

Let's start by defining the parameters of the dataset we're about to transform.

```
>>> n_feat = 128
>>> N = 20
>>> n_tasks = 2
```

Let's now make our dataset object

```
>>> import numpy as np
>>> import deepchem as dc
>>> X = np.random.randint(2, size=(N, n_feat))
>>> y = np.zeros((N, n_tasks))
>>> w = np.ones((N, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w)
```

And let's apply our transformer with 10 nearest neighbors.

```
>>> K = 10
>>> trans = dc.trans.IRVTransformer(K, n_tasks, dataset)
>>> dataset = trans.transform(dataset)
```

Note: This class requires TensorFlow to be installed.

__init__(*K: int*, *n_tasks: int*, *dataset:* Dataset)

Initializes IRVTransformer.

Parameters

- **K** (*int*) – number of nearest neighbours being count
- **n_tasks** (*int*) – number of tasks
- **dataset** (*dc.data.Dataset object*) – train_dataset

realize(*similarity: ndarray*, *y: ndarray*, *w: ndarray*) → List

find samples with top ten similarity values in the reference dataset

Parameters

- **similarity** (*np.ndarray*) – similarity value between target dataset and reference dataset should have size of (n_samples_in_target, n_samples_in_reference)
- **y** (*np.array*) – labels for a single task
- **w** (*np.array*) – weights for a single task

Returns

features – n_samples * np.array of size (2*K,) each array includes K similarity values and corresponding labels

Return type

list

X_transform(*X_target: ndarray*) → ndarray

Calculate similarity between target dataset(X_target) and

reference dataset(X): #(1 in intersection)/#(1 in union)

similarity = (X_target intersect X)/(X_target union X)

Parameters

X_target (*np.ndarray*) – fingerprints of target dataset should have same length with X in the second axis

Returns

X_target – features of size(batch_size, 2*K*n_tasks)

Return type

np.ndarray

static **matrix_mul**(*X1*, *X2*, *shard_size=5000*)

Calculate matrix multiplication for big matrix, X1 and X2 are sliced into pieces with shard_size rows(columns) then multiplied together and concatenated to the proper size

transform(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *DiskDataset* | *NumpyDataset*

Transforms a given dataset

Parameters

- **dataset** (Dataset) – Dataset to transform
- **parallel** (*bool, optional, (default False)*) – Whether to parallelize this transformation. Currently ignored.
- **out_dir** (*str, optional (default None)*) – Directory to write resulting dataset.

Returns

- *DiskDataset or NumpyDataset*
- *Dataset* object that is transformed.

untransform(*z: ndarray*) → ndarray

Not implemented.

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform_on_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

DAGTransformer

`class DAGTransformer`(*max_atoms: int = 50*)

Performs transform from ConvMol adjacency lists to DAG calculation orders

This transformer is used by *DAGModel* before training to transform its inputs to the correct shape. This expansion turns a molecule with *n* atoms into *n* DAGs, each with root at a different atom in the molecule.

Examples

Let's transform a small dataset of molecules.

```
>>> N = 10
>>> n_feat = 5
>>> import numpy as np
>>> feat = dc.feat.ConvMolFeaturizer()
>>> X = feat(["C", "CC"])
>>> y = np.random.rand(N)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> trans = dc.trans.DAGTransformer(max_atoms=5)
>>> dataset = trans.transform(dataset)
```

`__init__`(*max_atoms: int = 50*)

Initializes DAGTransformer.

Parameters

`max_atoms` (*`int, optional (Default 50)`*) – Maximum number of atoms to allow

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- **`X`** (*`np.ndarray`*) – Array of features
- **`y`** (*`np.ndarray`*) – Array of labels
- **`w`** (*`np.ndarray`*) – Array of weights.
- **`ids`** (*`np.ndarray`*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*z: ndarray*) → ndarray

Not implemented.

`UG_to_DAG`(*sample:* ConvMol) → List

This function generates the DAGs for a molecule

Parameters

`sample` (*ConvMol*) – Molecule to transform

Returns

List of parent adjacency matrices

Return type

List

transform(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns

A newly transformed Dataset object

Return type

Dataset

transform_on_array(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

RxnSplitTransformer

class RxnSplitTransformer(*sep_reagent: bool = True*, *dataset: Dataset | None = None*)

Splits the reaction SMILES input into the source and target strings required for machine translation tasks.

The input is expected to be in the form reactant>reagent>product. The source string would be reactants>reagents and the target string would be the products.

The transformer can also separate the reagents from the reactants for a mixed training mode. During mixed training, the source string is transformed from reactants>reagent to reactants.reagent> . This can be toggled (default True) by setting the value of sep_reagent while calling the transformer.

Examples

```
>>> # When mixed training is toggled.
>>> import numpy as np
>>> from deepchem.trans.transformers import RxnSplitTransformer
>>> reactions = np.array(["CC(C)C[Mg+].CON(C)C(=O)c1ccc(O)nc1>C1CCOC1.[Cl-]>
↪CC(C)CC(=O)c1ccc(O)nc1","CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(N)cc3)cc21.O=CO>>
↪CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(NC=O)cc3)cc21"], dtype=object)
>>> trans = RxnSplitTransformer(sep_reagent=True)
>>> split_reactions = trans.transform_array(X=reactions, y=np.array([]), w=np.
↪array([]), ids=np.array([]))
>>> split_reactions
(array([['CC(C)C[Mg+].CON(C)C(=O)c1ccc(O)nc1>C1CCOC1.[Cl-]',
        'CC(C)CC(=O)c1ccc(O)nc1'],
       ['CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(N)cc3)cc21.O=CO>',
        'CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(NC=O)cc3)cc21']], dtype='<U51'),␣
↪array([], dtype=float64), array([], dtype=float64), array([], dtype=float64))
```

When mixed training is disabled, you get the following outputs:

```
>>> trans_disable = RxnSplitTransformer(sep_reagent=False)
>>> split_reactions = trans_disable.transform_array(X=reactions, y=np.array([]),␣
↪w=np.array([]), ids=np.array([]))
>>> split_reactions
(array([['CC(C)C[Mg+].CON(C)C(=O)c1ccc(O)nc1.C1CCOC1.[Cl-]>',
        'CC(C)CC(=O)c1ccc(O)nc1'],
       ['CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(N)cc3)cc21.O=CO>',
        'CCn1cc(C(=O)O)c(=O)c2cc(F)c(-c3ccc(NC=O)cc3)cc21']], dtype='<U51'),␣
↪array([], dtype=float64), array([], dtype=float64), array([], dtype=float64))
```

Note: This class only transforms the feature field of a reaction dataset like USPTO.

`__init__`(*sep_reagent: bool = True*, *dataset: Dataset | None = None*)

Initializes the Reaction split Transformer.

Parameters

- **sep_reagent** (*bool, optional (default True)*) – To separate the reagent and reactants for training.
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed.

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features(the reactions)
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of weights.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform`(*dataset: Dataset*, *parallel: bool = False*, *out_dir: str | None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (`dc.data.Dataset`) – Dataset object to be transformed.
- **parallel** (`bool, optional (default False)`) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (`str, optional`) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns
A newly transformed Dataset object

Return type
Dataset

`transform_on_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (`np.ndarray`) – Array of features
- **y** (`np.ndarray`) – Array of labels
- **w** (`np.ndarray`) – Array of weights.
- **ids** (`np.ndarray`) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*z*)

Not Implemented.

3.12.3 Base Transformer (for develop)

The `dc.trans.Transformer` class is the abstract parent class for all transformers. This class should never be directly initialized, but contains a number of useful method implementations.

class Transformer(*transform_X: bool = False*, *transform_y: bool = False*, *transform_w: bool = False*, *transform_ids: bool = False*, *dataset:* Dataset | *None = None*)

Abstract base class for different data transformation techniques.

A transformer is an object that applies a transformation to a given dataset. Think of a transformation as a mathematical operation which makes the source dataset more amenable to learning. For example, one transformer could normalize the features for a dataset (ensuring they have zero mean and unit standard deviation). Another transformer could for example threshold values in a dataset so that values outside a given range are truncated. Yet another transformer could act as a data augmentation routine, generating multiple different images from each source datapoint (a transformation need not necessarily be one to one).

Transformers are designed to be chained, since data pipelines often chain multiple different transformations to a dataset. Transformers are also designed to be scalable and can be applied to large *dc.data.Dataset* objects. Not that Transformers are not usually thread-safe so you will have to be careful in processing very large datasets.

This class is an abstract superclass that isn't meant to be directly instantiated. Instead, you will want to instantiate one of the subclasses of this class inorder to perform concrete transformations.

__init__(*transform_X: bool = False*, *transform_y: bool = False*, *transform_w: bool = False*, *transform_ids: bool = False*, *dataset:* Dataset | *None = None*)

Initializes transformation based on dataset statistics.

Parameters

- **transform_X** (*bool, optional (default False)*) – Whether to transform X
- **transform_y** (*bool, optional (default False)*) – Whether to transform y
- **transform_w** (*bool, optional (default False)*) – Whether to transform w
- **transform_ids** (*bool, optional (default False)*) – Whether to transform ids
- **dataset** (*dc.data.Dataset object, optional (default None)*) – Dataset to be transformed

transform(*dataset:* Dataset, *parallel: bool = False*, *out_dir: str* | *None = None*, ***kwargs*) → *Dataset*

Transforms all internally stored data in dataset.

This method transforms all internal data in the provided dataset by using the *Dataset.transform* method. Note that this method adds X-transform, y-transform columns to metadata. Specified keyword arguments are passed on to *Dataset.transform*.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset object to be transformed.
- **parallel** (*bool, optional (default False)*) – if True, use multiple processes to transform the dataset in parallel. For large datasets, this might be faster.
- **out_dir** (*str, optional*) – If *out_dir* is specified in *kwargs* and *dataset* is a *DiskDataset*, the output dataset will be written to the specified directory.

Returns

A newly transformed Dataset object

Return type

Dataset

`transform_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transform the data in a set of (X, y, w, ids) arrays.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`transform_on_array`(*X: ndarray*, *y: ndarray*, *w: ndarray*, *ids: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Transforms numpy arrays X, y, and w

DEPRECATED. Use *transform_array* instead.

Parameters

- **X** (*np.ndarray*) – Array of features
- **y** (*np.ndarray*) – Array of labels
- **w** (*np.ndarray*) – Array of weights.
- **ids** (*np.ndarray*) – Array of identifiers.

Returns

- **Xtrans** (*np.ndarray*) – Transformed array of features
- **ytrans** (*np.ndarray*) – Transformed array of labels
- **wtrans** (*np.ndarray*) – Transformed array of weights
- **idstrans** (*np.ndarray*) – Transformed array of ids

`untransform`(*transformed: ndarray*) → ndarray

Reverses stored transformation on provided data.

Depending on whether *transform_X* or *transform_y* or *transform_w* was set, this will perform different un-transformations. Note that this method may not always be defined since some transformations aren't 1-1.

Parameters

transformed (*np.ndarray*) – Array which was previously transformed by this class.

3.13 Model Classes

DeepChem maintains an extensive collection of models for scientific applications. DeepChem's focus is on facilitating scientific applications, so we support a broad range of different machine learning frameworks (currently scikit-learn, xgboost, TensorFlow, and PyTorch) since different frameworks are more and less suited for different scientific applications.

3.13.1 Model Cheatsheet

If you're just getting started with DeepChem, you're probably interested in the basics. The place to get started is this "model cheatsheet" that lists various types of custom DeepChem models. Note that some wrappers like `SklearnModel` and `GBDTModel` which wrap external machine learning libraries are excluded, but this table should otherwise be complete.

As a note about how to read these tables: Each row describes what's needed to invoke a given model. Some models must be applied with given `Transformer` or `Featurizer` objects. Most models can be trained calling `model.fit`, otherwise the name of the fit_method is given in the Comment column. In order to run the models, make sure that the backend (Keras and tensorflow or Pytorch or Jax) is installed. You can thus read off what's needed to train the model from the table below.

General purpose

Table 2: General purpose models

Model	Reference	Classifier/Regre	Acceptable Featurizers	Back end	Comment
CNN		Classifier/ Regressor		Keras	
MultitaskClassifier		Classifier	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	PyTorch	
MultitaskFitTransformRegressor		Regressor	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	PyTorch	any `Transformer` can be used
MultitaskIRVClassifier		Classifier	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Keras	use `IRVTransformer`
MultitaskRegressor		Regressor	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Torch	
ProgressiveMultitaskClassifier	ref	Classifier	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Keras	
ProgressiveMultitaskRegressor	ref	Regressor	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Keras	
RobustMultitaskClassifier	ref	Classifier	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Keras	
RobustMultitaskRegressor	ref	Regressor	CircularFingerprint RDKitDescriptors CoulombMatrixEig RdkitGridFeaturizer BindingPocketFeaturizer ElementPropertyFingerprint	Keras	
SeqToSeq	ref			PyTorch	fit method: `fit_sequences`
WGAN	ref	Adversarial		Keras	fit method: `fit_gan`
UNet	ref	Classifier/ Regressor		PyTorch	

Molecules

Many models implemented in DeepChem were designed for small to medium-sized organic molecules, most often drug-like compounds. If your data is very different (e.g. molecules contain ‘exotic’ elements not present in the original dataset) or cannot be represented well using SMILES (e.g. metal complexes, crystals), some adaptations to the featurization and/or model might be needed to get reasonable results.

Table 3: Molecular models

Model	Reference	Type	Acceptable Featurizers	Backend	Comment
ScScoreModel	ref	Classifier	CircularFingerprint	Keras	
AtomicConvModel	ref	Classifier/ Regressor	ComplexNeighborListFragmentAtomicCoordinates	Keras	
AttentiveFPModel	ref	Classifier/ Regressor	MolGraphConvFeaturizer	PyTorch	
ChemCeption	ref	Classifier/ Regressor	SmilesToImage	Keras	
DAGModel	ref	Classifier/ Regressor	ConvMolFeaturizer	Keras	use DAGTransformer
GATModel	ref	Classifier/ Regressor	MolGraphConvFeaturizer	DGL/PyTorc	
GCNModel	ref	Classifier/ Regressor	MolGraphConvFeaturizer	DGL/PyTorc	
GraphConvModel	ref	Classifier/ Regressor	ConvMolFeaturizer	Keras	
MEGNetModel	ref	Classifier/ Regressor		PyTorch/PyTorc Geometric	
MPNNModel	ref	Classifier/ Regressor	MolGraphConvFeaturizer	DGL/PyTorc	
PagtnModel	ref	Classifier/ Regressor	PagtnMolGraphFeaturizer MolGraphConvFeaturizer	DGL/PyTorc	
Smiles2Vec	ref	Classifier/ Regressor	SmilesToSeq	Keras	
TextCNNModel	ref	Classifier/ Regressor		Keras/PyTorc	
DTNNModel	ref	Regressor	CoulombMatrix	PyTorch	
MATModel	ref	Regressor	MATFeaturizer	PyTorch	
Weave	ref	Regres	WeaveFeaturizer	Keras	

Materials

The following models were designed specifically for (inorganic) materials.

Table 4: Material models

Model	Reference	Type	Acceptable Featurizers	Backend	Comment
CGCNNModel	ref	Classifier/Regressor	CGCNNFEaturizer	DGL/PTorch	crystal graph CNN
MEGNetModel	ref	Classifier/Regressor		PyTorch/PyTorch Geometric	
LCNNModel	ref	Regressor	LCNNFeaturizer	PyTorch	lattice CNN

3.13.2 Model

class Model(*model=None*, *model_dir: str | None = None*, ***kwargs*)

Abstract base class for DeepChem models.

__init__(*model=None*, *model_dir: str | None = None*, ***kwargs*) → None

Abstract class for all models.

This is intended only for convenience of subclass implementations and should not be invoked directly.

Parameters

- **model** (*object*) – Wrapper around ScikitLearn/Keras/Tensorflow model object.
- **model_dir** (*str, optional (default None)*) – Path to directory where model will be stored. If not specified, model will be stored in a temporary directory.

fit_on_batch(*X: Sequence*, *y: Sequence*, *w: Sequence*)

Perform a single step of training.

Parameters

- **X** (*np.ndarray*) – the inputs for the batch
- **y** (*np.ndarray*) – the labels for the batch
- **w** (*np.ndarray*) – the weights for the batch

predict_on_batch(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*)

Makes predictions on given batch of new data.

Parameters

X (*np.ndarray*) – Features

reload() → None

Reload trained model from disk.

static get_model_filename(*model_dir: str*) → str

Given model directory, obtain filename for the model itself.

static get_params_filename(*model_dir: str*) → str

Given model directory, obtain filename for the model itself.

save() → None

Dispatcher function for saving.

Each subclass is responsible for overriding this method.

fit(*dataset:* Dataset)

Fits a model on data in a Dataset object.

Parameters

dataset (Dataset) – the Dataset to train on

predict(*dataset:* Dataset, *transformers: List[*Transformer*] = []*) → ndarray | Sequence[ndarray]

Uses self to make predictions on provided Dataset object.

Parameters

- **dataset** (Dataset) – Dataset to make prediction on
- **transformers** (*List[*Transformer*]*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.

Returns

A numpy array of predictions the model produces.

Return type

np.ndarray

evaluate(*dataset:* Dataset, *metrics: List[*Metric*]*, *transformers: List[*Transformer*] = []*, *per_task_metrics: bool = False*, *use_sample_weights: bool = False*, *n_classes: int = 2*)

Evaluates the performance of this model on specified dataset.

This function uses *Evaluator* under the hood to perform model evaluation. As a result, it inherits the same limitations of *Evaluator*. Namely, that only regression and classification models can be evaluated in this fashion. For generator models, you will need to overwrite this method to perform a custom evaluation.

Keyword arguments specified here will be passed to *Evaluator.compute_model_performance*.

Parameters

- **dataset** (Dataset) – Dataset object.
- **metrics** (*Metric / List[*Metric*] / function*) – The set of metrics provided. This class attempts to do some intelligent handling of input. If a single *dc.metrics.Metric* object is provided or a list is provided, it will evaluate *self.model* on these metrics. If a function is provided, it is assumed to be a metric function that this method will attempt to wrap in a *dc.metrics.Metric* object. A metric function must accept two arguments, *y_true, y_pred* both of which are *np.ndarray* objects and return a floating point score. The metric function may also accept a keyword argument *sample_weight* to account for per-sample weights.
- **transformers** (*List[*Transformer*]*) – List of *dc.trans.Transformer* objects. These transformations must have been applied to *dataset* previously. The dataset will be untransformed for metric evaluation.
- **per_task_metrics** (*bool, optional (default False)*) – If true, return computed metric for each task on multitask dataset.
- **use_sample_weights** (*bool, optional (default False)*) – If set, use per-sample weights *w*.

- **n_classes** (*int, optional (default None)*) – If specified, will use *n_classes* as the number of unique classes in *self.dataset*. Note that this argument will be ignored for regression metrics.

Returns

- **multitask_scores** (*dict*) – Dictionary mapping names of metrics to metric scores.
- **all_task_scores** (*dict, optional*) – If *per_task_metrics == True* is passed as a keyword argument, then returns a second dictionary of scores for each task separately.

get_task_type() → str

Currently models can only be classifiers or regressors.

get_num_tasks() → int

Get number of tasks.

3.14 Scikit-Learn Models

Scikit-learn's models can be wrapped so that they can interact conveniently with DeepChem. Oftentimes scikit-learn models are more robust and easier to train and are a nice first model to train.

3.14.1 SklearnModel

class **SklearnModel**(*model: BaseEstimator*, *model_dir: str | None = None*, ***kwargs*)

Wrapper class that wraps scikit-learn models as DeepChem models.

When you're working with scikit-learn and DeepChem, at times it can be useful to wrap a scikit-learn model as a DeepChem model. The reason for this might be that you want to do an apples-to-apples comparison of a scikit-learn model to another DeepChem model, or perhaps you want to use the hyperparameter tuning capabilities in *dc.hyper*. The *SklearnModel* class provides a wrapper around scikit-learn models that allows scikit-learn models to be trained on *Dataset* objects and evaluated with the same metrics as other DeepChem models.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> # Generating a random data and creating a dataset
>>> X, y = np.random.randn(5, 1), np.random.randn(5)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> # Wrapping a Sklearn Linear Regression model using DeepChem models API
>>> sklearn_model = LinearRegression()
>>> dc_model = dc.models.SklearnModel(sklearn_model)
>>> dc_model.fit(dataset)  # fitting dataset
```

Notes

All *SklearnModels* perform learning solely in memory. This means that it may not be possible to train *SklearnModel* on large `Dataset`s.

`__init__`(*model: BaseEstimator*, *model_dir: str | None = None*, ***kwargs*)

Parameters

- **`model`** (*`BaseEstimator`*) – The model instance which inherits a scikit-learn *BaseEstimator* Class.
- **`model_dir`** (*`str, optional (default None)`*) – If specified the model will be stored in this directory. Else, a temporary directory will be used.
- **`model_instance`** (*`BaseEstimator (DEPRECATED)`*) – The model instance which inherits a scikit-learn *BaseEstimator* Class.
- **`kwargs`** (*`dict`*) – kwargs['use_weights'] is a bool which determines if we pass weights into self.model.fit().

`fit`(*dataset:* Dataset) → None

Fits scikit-learn model to data.

Parameters
`dataset` (`Dataset`) – The *Dataset* to train this model on.

`predict_on_batch`(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Makes predictions on batch of data.

Parameters
`X` (*`np.ndarray`*) – A numpy array of features.

Returns
The value is a return value of *predict_proba* or *predict* method of the scikit-learn model. If the scikit-learn model has both methods, the value is always a return value of *predict_proba*.

Return type
np.ndarray

`predict`(*X:* Dataset, *transformers: List[*Transformer*] = []*) → ndarray | Sequence[ndarray]

Makes predictions on dataset.

Parameters

- **`dataset`** (`Dataset`) – Dataset to make prediction on.
- **`transformers`** (*`List[Transformer]`*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.

`save`()

Saves scikit-learn model to disk using joblib.

`reload`()

Loads scikit-learn model from joblib file on disk.

3.15 Gradient Boosting Models

Gradient Boosting Models (LightGBM and XGBoost) can be wrapped so they can interact with DeepChem.

3.15.1 GBDTModel

class GBDTModel(*model: BaseEstimator*, *model_dir: str | None = None*, *early_stopping_rounds: int = 50*, *eval_metric: Callable | str | None = None*, ***kwargs*)

Wrapper class that wraps GBDT models as DeepChem models.

This class supports LightGBM/XGBoost models.

__init__(*model: BaseEstimator*, *model_dir: str | None = None*, *early_stopping_rounds: int = 50*, *eval_metric: Callable | str | None = None*, ***kwargs*)

Parameters

- **model** (*BaseEstimator*) – The model instance of scikit-learn wrapper LightGBM/XGBoost models.
- **model_dir** (*str, optional (default None)*) – Path to directory where model will be stored.
- **early_stopping_rounds** (*int, optional (default 50)*) – Activates early stopping. Validation metric needs to improve at least once in every early_stopping_rounds round(s) to continue training.
- **eval_metric** (*Union[str, Callable]*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see official note for more details.

fit(*dataset: Dataset*)

Fits GDBT model with all data.

First, this function splits all data into train and valid data (8:2), and finds the best n_estimators. And then, we retrain all data using best n_estimators * 1.25.

Parameters

dataset (Dataset) – The *Dataset* to train this model on.

fit_with_eval(*train_dataset: Dataset*, *valid_dataset: Dataset*)

Fits GDBT model with valid data.

Parameters

- **train_dataset** (Dataset) – The *Dataset* to train this model on.
- **valid_dataset** (Dataset) – The *Dataset* to validate this model on.

3.16 Deep Learning Infrastructure

DeepChem maintains a lightweight layer of common deep learning model infrastructure that can be used for models built with different underlying frameworks. The losses and optimizers can be used for both TensorFlow and PyTorch models.

3.16.1 Losses

`class Loss`

A loss function for use in training models.

`class L1Loss`

The absolute difference between the true and predicted values.

`class HuberLoss`

Modified version of L1 Loss, also known as Smooth L1 loss. Less sensitive to small errors, linear for larger errors. Huber loss is generally better for cases where are are both large outliers as well as small, as compared to the L1 loss. By default, Delta = 1.0 and reduction = 'none'.

`class L2Loss`

The squared difference between the true and predicted values.

`class HingeLoss`

The hinge loss function.

The 'output' argument should contain logits, and all elements of 'labels' should equal 0 or 1.

`class SquaredHingeLoss`

The Squared Hinge loss function.

Defined as the square of the hinge loss between y_true and y_pred. The Squared Hinge Loss is differentiable.

`class PoissonLoss`

The Poisson loss function is defined as the mean of the elements of y_pred - (y_true * log(y_pred) for an input of (y_true, y_pred). Poisson loss is generally used for regression tasks where the data follows the poisson

`class BinaryCrossEntropy`

The cross entropy between pairs of probabilities.

The arguments should each have shape (batch_size) or (batch_size, tasks) and contain probabilities.

`class CategoricalCrossEntropy`

The cross entropy between two probability distributions.

The arguments should each have shape (batch_size, classes) or (batch_size, tasks, classes), and represent a probability distribution over classes.

`class SigmoidCrossEntropy`

The cross entropy between pairs of probabilities.

The arguments should each have shape (batch_size) or (batch_size, tasks). The labels should be probabilities, while the outputs should be logits that are converted to probabilities using a sigmoid function.

`class SoftmaxCrossEntropy`

The cross entropy between two probability distributions.

The arguments should each have shape (batch_size, classes) or (batch_size, tasks, classes). The labels should be probabilities, while the outputs should be logits that are converted to probabilities using a softmax function.

`class SparseSoftmaxCrossEntropy`

The cross entropy between two probability distributions.

The labels should have shape (batch_size) or (batch_size, tasks), and be integer class labels. The outputs have shape (batch_size, classes) or (batch_size, tasks, classes) and be logits that are converted to probabilities using a softmax function.

`class VAE_ELBO`

The Variational AutoEncoder loss, KL Divergence Regularize + marginal log-likelihood.

This losses based on _[1]. ELBO(Evidence lower bound) lexically replaced Variational lower bound. BCE means marginal log-likelihood, and KLD means KL divergence with normal distribution. Added hyper parameter 'kl_scale' for KLD.

The logvar and mu should have shape (batch_size, hidden_space). The x and reconstruction_x should have (batch_size, attribute). The kl_scale should be float.

Examples

Examples for calculating loss using constant tensor.

batch_size = 2, hidden_space = 2, num of original attribute = 3 >>> import numpy as np >>> import torch >>> import tensorflow as tf >>> logvar = np.array([[1.0,1.3],[0.6,1.2]]) >>> mu = np.array([[0.2,0.7],[1.2,0.4]]) >>> x = np.array([[0.9,0.4,0.8],[0.3,0,1]]) >>> reconstruction_x = np.array([[0.8,0.3,0.7],[0.2,0,0.9]])

Case tensorflow >>> VAE_ELBO()._compute_tf_loss(tf.constant(logvar), tf.constant(mu), tf.constant(x), tf.constant(reconstruction_x)) <tf.Tensor: shape=(2,), dtype=float64, numpy=array([0.70165154, 0.76238271])>

Case pytorch >>> (VAE_ELBO()._create_pytorch_loss())(torch.tensor(logvar), torch.tensor(mu), torch.tensor(x), torch.tensor(reconstruction_x)) tensor([0.7017, 0.7624], dtype=torch.float64)

References

`class VAE_KLDivergence`

The KL_divergence between hidden distribution and normal distribution.

This loss represents KL divergence losses between normal distribution(using parameter of distribution) based on _[1].

The logvar should have shape (batch_size, hidden_space) and each term represents standard deviation of hidden distribution. The mean shuold have (batch_size, hidden_space) and each term represents mean of hidden distribtuon.

Examples

Examples for calculating loss using constant tensor.

batch_size = 2, hidden_space = 2, >>> import numpy as np >>> import torch >>> import tensorflow as tf >>> logvar = np.array([[1.0,1.3],[0.6,1.2]]) >>> mu = np.array([[0.2,0.7],[1.2,0.4]])

Case tensorflow >>> VAE_KLDivergence()._compute_tf_loss(tf.constant(logvar), tf.constant(mu)) <tf.Tensor: shape=(2,), dtype=float64, numpy=array([0.17381787, 0.51425203])>

Case pytorch >>> (VAE_KLDivergence()._create_pytorch_loss())(torch.tensor(logvar), torch.tensor(mu)) tensor([0.1738, 0.5143], dtype=torch.float64)

References

`class ShannonEntropy`

The ShannonEntropy of discrete-distribution.

This loss represents shannon entropy based on _[1].

The inputs should have shape (batch size, num of variable) and represents probabilites distribution.

Examples

Examples for calculating loss using constant tensor.

batch_size = 2, num_of variable = variable, >>> import numpy as np >>> import torch >>> import tensorflow as tf >>> inputs = np.array([[0.7,0.3],[0.9,0.1]])

Case tensorflow >>> ShannonEntropy()._compute_tf_loss(tf.constant(inputs)) <tf.Tensor: shape=(2,), dtype=float64, numpy=array([0.30543215, 0.16254149])>

Case pytorch >>> (ShannonEntropy()._create_pytorch_loss())(torch.tensor(inputs)) tensor([0.3054, 0.1625], dtype=torch.float64)

References

`class GlobalMutualInformationLoss`

Global-global encoding loss (comparing two full graphs).

Compares the encodings of two molecular graphs and returns the loss between them based on the measure specified. The encodings are generated by two separate encoders in order to maximize the mutual information between the two encodings.

Parameters

- **global_enc** (*torch.Tensor*) – Features from a graph convolutional encoder.
- **global_enc2** (*torch.Tensor*) – Another set of features from a graph convolutional encoder.
- **measure** (*str*) – The divergence measure to use for the unsupervised loss. Options are 'GAN', 'JSD', 'KL', 'RKL', 'X2', 'DV', 'H2', or 'W1'.
- **average_loss** (*bool*) – Whether to average the loss over the batch

Returns

loss – Measure of mutual information between the encodings of the two graphs.

Return type

torch.Tensor

References

Examples

```
>>> import numpy as np
>>> import deepchem.models.losses as losses
>>> from deepchem.feat.graph_data import BatchGraphData, GraphData
>>> from deepchem.models.torch_models.infograph import InfoGraphEncoder
>>> from deepchem.models.torch_models.layers import MultilayerPerceptron
>>> graph_list = []
>>> for i in range(3):
...     node_features = np.random.rand(5, 10)
...     edge_index = np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]], dtype=np.int64)
...     edge_features = np.random.rand(5, 5)
...     graph_list.append(GraphData(node_features, edge_index, edge_features))
>>> batch = BatchGraphData(graph_list).numpy_to_torch()
>>> num_feat = 10
>>> edge_dim = 5
>>> dim = 4
>>> encoder = InfoGraphEncoder(num_feat, edge_dim, dim)
>>> encoding, feature_map = encoder(batch)
>>> g_enc = MultilayerPerceptron(2 * dim, dim)(encoding)
>>> g_enc2 = MultilayerPerceptron(2 * dim, dim)(encoding)
>>> globalloss = losses.GlobalMutualInformationLoss()
>>> loss = globalloss._create_pytorch_loss()(g_enc, g_enc2).detach().numpy()
```

class LocalMutualInformationLoss

Local-global encoding loss (comparing a subgraph to the full graph).

Compares the encodings of two molecular graphs and returns the loss between them based on the measure specified. The encodings are generated by two separate encoders in order to maximize the mutual information between the two encodings.

Parameters

- **local_enc** (*torch.Tensor*) – Features from a graph convolutional encoder.
- **global_enc** (*torch.Tensor*) – Another set of features from a graph convolutional encoder.
- **batch_graph_index** (*graph_index: np.ndarray or torch.tensor, dtype int*) – This vector indicates which graph the node belongs with shape [num_nodes,]. Only present in BatchGraphData, not in GraphData objects.
- **measure** (*str*) – The divergence measure to use for the unsupervised loss. Options are 'GAN', 'JSD', 'KL', 'RKL', 'X2', 'DV', 'H2', or 'W1'.
- **average_loss** (*bool*) – Whether to average the loss over the batch

Returns

loss – Measure of mutual information between the encodings of the two graphs.

Return type

torch.Tensor

References

Example

```
>>> import numpy as np
>>> import deepchem.models.losses as losses
>>> from deepchem.feat.graph_data import BatchGraphData, GraphData
>>> from deepchem.models.torch_models.infograph import InfoGraphEncoder
>>> from deepchem.models.torch_models.layers import MultilayerPerceptron
>>> graph_list = []
>>> for i in range(3):
...     node_features = np.random.rand(5, 10)
...     edge_index = np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]], dtype=np.int64)
...     edge_features = np.random.rand(5, 5)
...     graph_list.append(GraphData(node_features, edge_index, edge_features))
```

```
>>> batch = BatchGraphData(graph_list).numpy_to_torch()
>>> num_feat = 10
>>> edge_dim = 5
>>> dim = 4
>>> encoder = InfoGraphEncoder(num_feat, edge_dim, dim)
>>> encoding, feature_map = encoder(batch)
>>> g_enc = MultilayerPerceptron(2 * dim, dim)(encoding)
>>> l_enc = MultilayerPerceptron(dim, dim)(feature_map)
>>> localloss = losses.LocalMutualInformationLoss()
>>> loss = localloss._create_pytorch_loss()(l_enc, g_enc, batch.graph_index).
↪detach().numpy()
```

class GroverPretrainLoss

The Grover Pretraining consists learning of atom embeddings and bond embeddings for a molecule. To this end, the learning consists of three tasks:

1. Learning of atom vocabulary from atom embeddings and bond embeddings
2. Learning of bond vocabulary from atom embeddings and bond embeddings
3. Learning to predict functional groups from atom embedings readout and bond embeddings readout

The loss function accepts atom vocabulary labels, bond vocabulary labels and functional group predictions produced by Grover model during pretraining as a dictionary and applies negative log-likelihood loss for atom vocabulary and bond vocabulary predictions and Binary Cross Entropy loss for functional group prediction and sums these to get overall loss.

Example

```
>>> import torch
>>> from deepchem.models.losses import GroverPretrainLoss
>>> loss = GroverPretrainLoss()
>>> loss_fn = loss._create_pytorch_loss()
>>> batch_size = 3
>>> output_dim = 10
>>> fg_size = 8
>>> atom_vocab_task_target = torch.ones(batch_size).type(torch.int64)
```

(continues on next page)

(continued from previous page)

```
>>> bond_vocab_task_target = torch.ones(batch_size).type(torch.int64)
>>> fg_task_target = torch.ones(batch_size, fg_size)
>>> atom_vocab_task_atom_pred = torch.zeros(batch_size, output_dim)
>>> bond_vocab_task_atom_pred = torch.zeros(batch_size, output_dim)
>>> atom_vocab_task_bond_pred = torch.zeros(batch_size, output_dim)
>>> bond_vocab_task_bond_pred = torch.zeros(batch_size, output_dim)
>>> fg_task_atom_from_atom = torch.zeros(batch_size, fg_size)
>>> fg_task_atom_from_bond = torch.zeros(batch_size, fg_size)
>>> fg_task_bond_from_atom = torch.zeros(batch_size, fg_size)
>>> fg_task_bond_from_bond = torch.zeros(batch_size, fg_size)
>>> result = loss_fn(atom_vocab_task_atom_pred, atom_vocab_task_bond_pred,
...     bond_vocab_task_atom_pred, bond_vocab_task_bond_pred, fg_task_atom_from_
↪atom,
...     fg_task_atom_from_bond, fg_task_bond_from_atom, fg_task_bond_from_bond,
...     atom_vocab_task_target, bond_vocab_task_target, fg_task_target)
```

Reference

class EdgePredictionLoss

EdgePredictionLoss is an unsupervised graph edge prediction loss function that calculates the loss based on the similarity between node embeddings for positive and negative edge pairs. This loss function is designed for graph neural networks and is particularly useful for pre-training tasks.

This loss function encourages the model to learn node embeddings that can effectively distinguish between true edges (positive samples) and false edges (negative samples) in the graph.

The loss is computed by comparing the similarity scores (dot product) of node embeddings for positive and negative edge pairs. The goal is to maximize the similarity for positive pairs and minimize it for negative pairs.

To use this loss function, the input must be a BatchGraphData object transformed by the negative_edge_sampler. The loss function takes the node embeddings and the input graph data (with positive and negative edge pairs) as inputs and returns the edge prediction loss.

Examples

```
>>> from deepchem.models.losses import EdgePredictionLoss
>>> from deepchem.feat.graph_data import BatchGraphData, GraphData
>>> from deepchem.models.torch_models.gnn import negative_edge_sampler
>>> import torch
>>> import numpy as np
>>> emb_dim = 8
>>> num_nodes_list, num_edge_list = [3, 4, 5], [2, 4, 5]
>>> num_node_features, num_edge_features = 32, 32
>>> edge_index_list = [
...     np.array([[0, 1], [1, 2]]),
...     np.array([[0, 1, 2, 3], [1, 2, 0, 2]]),
...     np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]),
... ]
>>> graph_list = [
...     GraphData(node_features=np.random.random_sample(
...         (num_nodes_list[i], num_node_features)),
```

(continues on next page)

(continued from previous page)

```
...                 edge_index=edge_index_list[i],
...                 edge_features=np.random.random_sample(
...                     (num_edge_list[i], num_edge_features)),
...                 node_pos_features=None) for i in range(len(num_edge_list))
... ]
>>> batched_graph = BatchGraphData(graph_list)
>>> batched_graph = batched_graph.numpy_to_torch()
>>> neg_sampled = negative_edge_sampler(batched_graph)
>>> embedding = np.random.random((sum(num_nodes_list), emb_dim))
>>> embedding = torch.from_numpy(embedding)
>>> loss_func = EdgePredictionLoss()._create_pytorch_loss()
>>> loss = loss_func(embedding, neg_sampled)
```

References

class GraphNodeMaskingLoss

GraphNodeMaskingLoss is an unsupervised graph node masking loss function that calculates the loss based on the predicted node labels and true node labels. This loss function is designed for graph neural networks and is particularly useful for pre-training tasks.

This loss function encourages the model to learn node embeddings that can effectively predict the masked node labels in the graph.

The loss is computed using the CrossEntropyLoss between the predicted node labels and the true node labels.

To use this loss function, the input must be a BatchGraphData object transformed by the mask_nodes function. The loss function takes the predicted node labels, predicted edge labels, and the input graph data (with masked node labels) as inputs and returns the node masking loss.

Parameters

- **pred_node** (*torch.Tensor*) – Predicted node labels
- **pred_edge** (*Optional(torch.Tensor)*) – Predicted edge labels
- **inputs** (*BatchGraphData*) – Input graph data with masked node and edge labels

Examples

```
>>> from deepchem.models.losses import GraphNodeMaskingLoss
>>> from deepchem.feat.graph_data import BatchGraphData, GraphData
>>> from deepchem.models.torch_models.gnn import mask_nodes
>>> import torch
>>> import numpy as np
>>> num_nodes_list, num_edge_list = [3, 4, 5], [2, 4, 5]
>>> num_node_features, num_edge_features = 32, 32
>>> edge_index_list = [
...     np.array([[0, 1], [1, 2]]),
...     np.array([[0, 1, 2, 3], [1, 2, 0, 2]]),
...     np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]),
... ]
>>> graph_list = [
...     GraphData(node_features=np.random.random_sample(
```

(continues on next page)

(continued from previous page)

```
...         (num_nodes_list[i], num_node_features)),
...               edge_index=edge_index_list[i],
...               edge_features=np.random.random_sample(
...                   (num_edge_list[i], num_edge_features)),
...               node_pos_features=None) for i in range(len(num_edge_list))
... ]
>>> batched_graph = BatchGraphData(graph_list)
>>> batched_graph = batched_graph.numpy_to_torch()
>>> masked_graph = mask_nodes(batched_graph, 0.1)
>>> pred_node = torch.randn((sum(num_nodes_list), num_node_features))
>>> pred_edge = torch.randn((sum(num_edge_list), num_edge_features))
>>> loss_func = GraphNodeMaskingLoss()._create_pytorch_loss()
>>> loss = loss_func(pred_node[masked_graph.masked_node_indices],
...                  pred_edge[masked_graph.connected_edge_indices], masked_graph)
```

References

class GraphEdgeMaskingLoss

GraphEdgeMaskingLoss is an unsupervised graph edge masking loss function that calculates the loss based on the predicted edge labels and true edge labels. This loss function is designed for graph neural networks and is particularly useful for pre-training tasks.

This loss function encourages the model to learn node embeddings that can effectively predict the masked edge labels in the graph.

The loss is computed using the CrossEntropyLoss between the predicted edge labels and the true edge labels.

To use this loss function, the input must be a BatchGraphData object transformed by the mask_edges function. The loss function takes the predicted edge labels and the true edge labels as inputs and returns the edge masking loss.

Parameters

- **pred_edge** (*torch.Tensor*) – Predicted edge labels.
- **inputs** (*BatchGraphData*) – Input graph data (with masked edge labels).

Examples

```
>>> from deepchem.models.losses import GraphEdgeMaskingLoss
>>> from deepchem.feat.graph_data import BatchGraphData, GraphData
>>> from deepchem.models.torch_models.gnn import mask_edges
>>> import torch
>>> import numpy as np
>>> num_nodes_list, num_edge_list = [3, 4, 5], [2, 4, 5]
>>> num_node_features, num_edge_features = 32, 32
>>> edge_index_list = [
...     np.array([[0, 1], [1, 2]]),
...     np.array([[0, 1, 2, 3], [1, 2, 0, 2]]),
...     np.array([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]),
... ]
>>> graph_list = [
...     GraphData(node_features=np.random.random_sample(
```

(continues on next page)

(continued from previous page)

```
...         (num_nodes_list[i], num_node_features)),
...             edge_index=edge_index_list[i],
...             edge_features=np.random.random_sample(
...                 (num_edge_list[i], num_edge_features)),
...             node_pos_features=None) for i in range(len(num_edge_list))
... ]
>>> batched_graph = BatchGraphData(graph_list)
>>> batched_graph = batched_graph.numpy_to_torch()
>>> masked_graph = mask_edges(batched_graph, .1)
>>> pred_edge = torch.randn((sum(num_edge_list), num_edge_features))
>>> loss_func = GraphEdgeMaskingLoss()._create_pytorch_loss()
>>> loss = loss_func(pred_edge[masked_graph.masked_edge_idx], masked_graph)
```

References

class DeepGraphInfomaxLoss

Loss that maximizes mutual information between local node representations and a pooled global graph representation. This is to encourage nearby nodes to have similar embeddings.

Parameters

- **positive_score** (*torch.Tensor*) – Positive score. This score measures the similarity between the local node embeddings (*node_emb*) and the global graph representation (*positive_expanded_summary_emb*) derived from the same graph. The goal is to maximize this score, as it indicates that the local node embeddings and the global graph representation are highly correlated, capturing the mutual information between them.
- **negative_score** (*torch.Tensor*) – Negative score. This score measures the similarity between the local node embeddings (*node_emb*) and the global graph representation (*negative_expanded_summary_emb*) derived from a different graph (shifted by one position in this case). The goal is to minimize this score, as it indicates that the local node embeddings and the global graph representation from different graphs are not correlated, ensuring that the model learns meaningful representations that are specific to each graph.

Examples

```
>>> import torch
>>> import numpy as np
>>> from deepchem.feat.graph_data import GraphData
>>> from torch_geometric.nn import global_mean_pool
>>> from deepchem.models.losses import DeepGraphInfomaxLoss
>>> x = np.array([[1, 0], [0, 1], [1, 1], [0, 0]])
>>> edge_index = np.array([[0, 1, 2, 0, 3], [1, 0, 1, 3, 2]])
>>> graph_index = np.array([0, 0, 1, 1])
>>> data = GraphData(node_features=x, edge_index=edge_index, graph_index=graph_
↪index).numpy_to_torch()
>>> graph_infomax_loss = DeepGraphInfomaxLoss()._create_pytorch_loss()
>>> # Initialize node_emb randomly
>>> num_nodes = data.num_nodes
>>> embedding_dim = 8
>>> node_emb = torch.randn(num_nodes, embedding_dim)
```

(continues on next page)

(continued from previous page)

```
>>> # Compute the global graph representation
>>> summary_emb = global_mean_pool(node_emb, data.graph_index)
>>> # Compute positive and negative scores
>>> positive_score = torch.matmul(node_emb, summary_emb.t())
>>> negative_score = torch.matmul(node_emb, summary_emb.roll(1, dims=0).t())
>>> loss = graph_infomax_loss(positive_score, negative_score)
```

References

class GraphContextPredLoss

GraphContextPredLoss is a loss function designed for graph neural networks that aims to predict the context of a node given its substructure. The context of a node is essentially the ring of nodes around it outside of an inner k1-hop diameter and inside an outer k2-hop diameter.

This loss compares the representation of a node's neighborhood with the representation of the node's context. It then uses negative sampling to compare the representation of the node's neighborhood with the representation of a random node's context.

Parameters

- **mode** (*str*) – The mode of the model. It can be either "cbow" (continuous bag of words) or "skipgram".
- **neg_samples** (*int*) – The number of negative samples to use for negative sampling.

Examples

```
>>> import torch
>>> from deepchem.models.losses import GraphContextPredLoss
>>> substruct_rep = torch.randn(4, 8)
>>> overlapped_node_rep = torch.randn(8, 8)
>>> context_rep = torch.randn(4, 8)
>>> neg_context_rep = torch.randn(2 * 4, 8)
>>> overlapped_context_size = torch.tensor([2, 2, 2, 2])
>>> mode = "cbow"
>>> neg_samples = 2
>>> graph_context_pred_loss = GraphContextPredLoss()._create_pytorch_loss(mode, neg_
↪samples)
>>> loss = graph_context_pred_loss(substruct_rep, overlapped_node_rep, context_rep,␣
↪neg_context_rep, overlapped_context_size)
```

class DensityProfileLoss

Loss for the density profile entry type for Quantum Chemistry calculations. It is an integration of the squared difference between ground truth and calculated values, at all spaces in the integration grid.

Examples

```
>>> from deepchem.models.losses import DensityProfileLoss
>>> import torch
>>> volume = torch.Tensor([2.0])
>>> output = torch.Tensor([3.0])
>>> labels = torch.Tensor([4.0])
>>> loss = (DensityProfileLoss()._create_pytorch_loss(volume))(output, labels)
>>> # Generating volume tensor for an entry object:
>>> from deepchem.feat.dft_data import DFTEntry
>>> e_type = 'dens'
>>> true_val = 0
>>> systems =[{'moldesc': 'H 0.86625 0 0; F -0.86625 0 0','basis' : '6-311++G(3df,
→3pd)'}]
>>> dens_entry_for_HF = DFTEntry.create(e_type, true_val, systems)
>>> grid = (dens_entry_for_HF).get_integration_grid()
```

The 6-311++G(3df,3pd) basis for atomz 1 does not exist, but we will download it Downloaded to /usr/share/miniconda3/envs/deepchem/lib/python3.8/site-packages/dqc/api/.database/**6-311ppg_3df_3pd_**/01.gaussian94 The 6-311++G(3df,3pd) basis for atomz 9 does not exist, but we will download it Downloaded to /usr/share/miniconda3/envs/deepchem/lib/python3.8/site-packages/dqc/api/.database/**6-311ppg_3df_3pd_**/09.gaussian94

```
>>> volume = grid.get_dvolume()
```

References

Kasim, Muhammad F., and Sam M. Vinko. "Learning the exchange-correlation functional from nature with fully differentiable density functional theory." Physical Review Letters 127.12 (2021): 126403. https://github.com/deepchem/deepchem/blob/0bc3139bb99ae7700ba2325a6756e33b6c327842/deepchem/models/dft/dftxc.py

class NTXentMultiplePositives(*norm: bool = True*, *tau: float = 0.5*, *uniformity_reg=0*, *variance_reg=0*, *covariance_reg=0*, *conformer_variance_reg=0*)

This is a modification of the NTXent loss function from Chen [1]_. This loss is designed for contrastive learning of molecular representations, comparing the similarity of a molecule's latent representation to positive and negative samples.

The modifications proposed in [2]_ enable multiple conformers to be used as positive samples.

This loss function is designed for graph neural networks and is particularly useful for unsupervised pre-training tasks.

Parameters

- **norm** (*bool, optional (default=True)*) – Whether to normalize the similarity matrix.
- **tau** (*float, optional (default=0.5)*) – Temperature parameter for the similarity matrix.
- **uniformity_reg** (*float, optional (default=0)*) – Regularization weight for the uniformity loss.
- **variance_reg** (*float, optional (default=0)*) – Regularization weight for the variance loss.
- **covariance_reg** (*float, optional (default=0)*) – Regularization weight for the covariance loss.

- **conformer_variance_reg** (*float, optional (default=0)*) – Regularization weight for the conformer variance loss.

Examples

```
>>> import torch
>>> from deepchem.models.losses import NTXentMultiplePositives
>>> z1 = torch.randn(4, 8)
>>> z2 = torch.randn(4 * 3, 8)
>>> ntxent_loss = NTXentMultiplePositives(norm=True, tau=0.5)
>>> loss_fn = ntxent_loss._create_pytorch_loss()
>>> loss = loss_fn(z1, z2)
```

References

__init__(*norm: bool = True*, *tau: float = 0.5*, *uniformity_reg=0*, *variance_reg=0*, *covariance_reg=0*, *conformer_variance_reg=0*) → None

3.16.2 Optimizers

class Optimizer(*learning_rate: float | LearningRateSchedule*)

An algorithm for optimizing a model.

This is an abstract class. Subclasses represent specific optimization algorithms.

__init__(*learning_rate: float | LearningRateSchedule*)

This constructor should only be called by subclasses.

Parameters

learning_rate (*float or LearningRateSchedule*) – the learning rate to use for optimization

class LearningRateSchedule

A schedule for changing the learning rate over the course of optimization.

This is an abstract class. Subclasses represent specific schedules.

class AdaGrad(*learning_rate: float | LearningRateSchedule = 0.001*, *initial_accumulator_value: float = 0.1*, *epsilon: float = 1e-07*)

The AdaGrad optimization algorithm.

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates. See [1]_ for a full reference for the algorithm.

References

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*, *initial_accumulator_value: float = 0.1*, *epsilon: float = 1e-07*)

Construct an AdaGrad optimizer. :param learning_rate: the learning rate to use for optimization :type learning_rate: float or LearningRateSchedule :param initial_accumulator_value: a parameter of the AdaGrad algorithm :type initial_accumulator_value: float :param epsilon: a parameter of the AdaGrad algorithm :type epsilon: float

class **Adam**(*learning_rate: float* | LearningRateSchedule *= 0.001*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*, *weight_decay: float = 0*)

The Adam optimization algorithm.

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*, *weight_decay: float = 0*)

Construct an Adam optimizer.

Parameters

- **learning_rate** (*float or* LearningRateSchedule) – the learning rate to use for optimization
- **beta1** (*float*) – a parameter of the Adam algorithm
- **beta2** (*float*) – a parameter of the Adam algorithm
- **epsilon** (*float*) – a parameter of the Adam algorithm
- **weight_decay** (*float*) – L2 penalty - a parameter of the Adam algorithm

class **AdamW**(*learning_rate: float* | LearningRateSchedule *= 0.001*, *weight_decay: float* | LearningRateSchedule *= 0.01*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*, *amsgrad: bool = False*)

The AdamW optimization algorithm. AdamW is a variant of Adam, with improved weight decay. In Adam, weight decay is implemented as: weight_decay (float, optional) – weight decay (L2 penalty) (default: 0) In AdamW, weight decay is implemented as: weight_decay (float, optional) – weight decay coefficient (default: 1e-2)

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*, *weight_decay: float* | LearningRateSchedule *= 0.01*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*, *amsgrad: bool = False*)

Construct an AdamW optimizer. :param learning_rate: the learning rate to use for optimization :type learning_rate: float or LearningRateSchedule :param weight_decay: weight decay coefficient for AdamW :type weight_decay: float or LearningRateSchedule :param beta1: a parameter of the Adam algorithm :type beta1: float :param beta2: a parameter of the Adam algorithm :type beta2: float :param epsilon: a parameter of the Adam algorithm :type epsilon: float :param amsgrad: If True, will use the AMSGrad variant of AdamW (from "On the Convergence of Adam and Beyond"), else will use the original algorithm. :type amsgrad: bool

class **SparseAdam**(*learning_rate: float* | LearningRateSchedule *= 0.001*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*)

The Sparse Adam optimization algorithm, also known as Lazy Adam. Sparse Adam is suitable for sparse tensors. It handles sparse updates more efficiently. It only updates moving-average accumulators for sparse variable indices that appear in the current batch, rather than updating the accumulators for all indices.

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*)

Construct an Adam optimizer.

Parameters

- **learning_rate** (*float or* LearningRateSchedule) – the learning rate to use for optimization
- **beta1** (*float*) – a parameter of the SparseAdam algorithm
- **beta2** (*float*) – a parameter of the SparseAdam algorithm
- **epsilon** (*float*) – a parameter of the SparseAdam algorithm

class RMSProp(*learning_rate: float* | LearningRateSchedule *= 0.001*, *momentum: float = 0.0*, *decay: float = 0.9*, *epsilon: float = 1e-10*)

RMSProp Optimization algorithm.

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*, *momentum: float = 0.0*, *decay: float = 0.9*, *epsilon: float = 1e-10*)

Construct an RMSProp Optimizer.

Parameters

- **learning_rate** (*float or* LearningRateSchedule) – the learning_rate used for optimization
- **momentum** (*float, default 0.0*) – a parameter of the RMSProp algorithm
- **decay** (*float, default 0.9*) – a parameter of the RMSProp algorithm
- **epsilon** (*float, default 1e-10*) – a parameter of the RMSProp algorithm

class GradientDescent(*learning_rate: float* | LearningRateSchedule *= 0.001*)

The gradient descent optimization algorithm.

__init__(*learning_rate: float* | LearningRateSchedule *= 0.001*)

Construct a gradient descent optimizer.

Parameters

learning_rate (*float or* LearningRateSchedule) – the learning rate to use for optimization

class ExponentialDecay(*initial_rate: float*, *decay_rate: float*, *decay_steps: int*, *staircase: bool = True*)

A learning rate that decreases exponentially with the number of training steps.

__init__(*initial_rate: float*, *decay_rate: float*, *decay_steps: int*, *staircase: bool = True*)

Create an exponentially decaying learning rate.

The learning rate starts as initial_rate. Every decay_steps training steps, it is multiplied by decay_rate.

Parameters

- **initial_rate** (*float*) – the initial learning rate
- **decay_rate** (*float*) – the base of the exponential
- **decay_steps** (*int*) – the number of training steps over which the rate decreases by decay_rate
- **staircase** (*bool*) – if True, the learning rate decreases by discrete jumps every decay_steps. if False, the learning rate decreases smoothly every step

class PolynomialDecay(*initial_rate: float*, *final_rate: float*, *decay_steps: int*, *power: float = 1.0*)

A learning rate that decreases from an initial value to a final value over a fixed number of training steps.

`__init__`(*initial_rate: float*, *final_rate: float*, *decay_steps: int*, *power: float = 1.0*)

Create a smoothly decaying learning rate.

The learning rate starts as initial_rate. It smoothly decreases to final_rate over decay_steps training steps. It decays as a function of (1-step/decay_steps)**power. Once the final rate is reached, it remains there for the rest of optimization.

Parameters

- **initial_rate** (*float*) – the initial learning rate
- **final_rate** (*float*) – the final learning rate
- **decay_steps** (*int*) – the number of training steps over which the rate decreases from initial_rate to final_rate
- **power** (*float*) – the exponent controlling the shape of the decay

class LinearCosineDecay(*initial_rate: float*, *decay_steps: int*, *alpha: float = 0.0*, *beta: float = 0.001*, *num_periods: float = 0.5*)

Applies linear cosine decay to the learning rate

`__init__`(*initial_rate: float*, *decay_steps: int*, *alpha: float = 0.0*, *beta: float = 0.001*, *num_periods: float = 0.5*)

Parameters

- **learning_rate** (*float*) –
- **rate** (*initial learning*) –
- **decay_steps** (*int*) –
- **over** (*number of steps to decay*) –
- **num_periods** (*number of periods in the cosine part of the decay*) –

3.17 Keras Models

DeepChem extensively uses Keras to build deep learning models.

3.17.1 KerasModel

Training loss and validation metrics can be automatically logged to Weights & Biases with the following commands:

```
# Install wandb in shell
pip install wandb

# Login in shell (required only once)
wandb login
# Login in notebook (required only once)
import wandb
wandb.login()

# Initialize a WandbLogger
logger = WandbLogger(...)

```

(continues on next page)

(continued from previous page)

```
# Set `wandb_logger` when creating `KerasModel`
import deepchem as dc
# Log training loss to wandb
model = dc.models.KerasModel(..., wandb_logger=logger)
model.fit(...)

# Log validation metrics to wandb using ValidationCallback
import deepchem as dc
vc = dc.models.ValidationCallback(...)
model = KerasModel(..., wandb_logger=logger)
model.fit(..., callbacks=[vc])
logger.finish()
```

`class KerasModel`(*model: Model*, *loss: Loss | Callable[[List, List, List], Any]*, *output_types: List[str] | None = None*, *batch_size: int = 100*, *model_dir: str | None = None*, *learning_rate: float | LearningRateSchedule = 0.001*, *optimizer: Optimizer | None = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *wandb_logger: WandbLogger | None = None*, ***kwargs*)

This is a DeepChem model implemented by a Keras model.

This class provides several advantages over using the Keras model's fitting and prediction methods directly.

1. **It provides better integration with the rest of DeepChem,**
   such as direct support for Datasets and Transformers.
2. **It defines the loss in a more flexible way. In particular,**
   Keras does not support multidimensional weight matrices, which makes it impossible to implement most multitask models with Keras.
3. **It provides various additional features not found in the**
   Keras model class, such as uncertainty prediction and saliency mapping.

Here is a simple example of code that uses KerasModel to train a Keras model on a DeepChem dataset.

>> keras_model = tf.keras.Sequential([>> tf.keras.layers.Dense(1000, activation='tanh'), >> tf.keras.layers.Dense(1) >>]) >> model = KerasModel(keras_model, loss=dc.models.losses.L2Loss()) >> model.fit(dataset)

The loss function for a model can be defined in two different ways. For models that have only a single output and use a standard loss function, you can simply provide a dc.models.losses.Loss object. This defines the loss for each sample or sample/task pair. The result is automatically multiplied by the weights and averaged over the batch. Any additional losses computed by model layers, such as weight decay penalties, are also added.

For more complicated cases, you can instead provide a function that directly computes the total loss. It must be of the form f(outputs, labels, weights), taking the list of outputs from the model, the expected values, and any weight matrices. It should return a scalar equal to the value of the loss function for the batch. No additional processing is done to the result; it is up to you to do any weighting, averaging, adding of penalty terms, etc.

You can optionally provide an output_types argument, which describes how to interpret the model's outputs. This should be a list of strings, one for each output. You can use an arbitrary output_type for a output, but some output_types are special and will undergo extra processing:

- **'prediction': This is a normal output, and will be returned by predict().**
  If output types are not specified, all outputs are assumed to be of this type.
- **'loss': This output will be used in place of the normal**
  outputs for computing the loss function. For example, models that output probability distributions usually do it by computing unbounded numbers (the logits), then passing them through a softmax

function to turn them into probabilities. When computing the cross entropy, it is more numerically stable to use the logits directly rather than the probabilities. You can do this by having the model produce both probabilities and logits as outputs, then specifying output_types=['prediction', 'loss']. When predict() is called, only the first output (the probabilities) will be returned. But during training, it is the second output (the logits) that will be passed to the loss function.

- **'variance': This output is used for estimating the**
  uncertainty in another output. To create a model that can estimate uncertainty, there must be the same number of 'prediction' and 'variance' outputs. Each variance output must have the same shape as the corresponding prediction output, and each element is an estimate of the variance in the corresponding prediction. Also be aware that if a model supports uncertainty, it MUST use dropout on every layer, and dropout most be enabled during uncertainty prediction. Otherwise, the uncertainties it computes will be inaccurate.
- **other: Arbitrary output_types can be used to extract outputs**
  produced by the model, but will have no additional processing performed.

`__init__`(*model: Model*, *loss: Loss | Callable[[List, List, List], Any]*, *output_types: List[str] | None = None*, *batch_size: int = 100*, *model_dir: str | None = None*, *learning_rate: float | LearningRateSchedule = 0.001*, *optimizer: Optimizer | None = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *wandb_logger: WandbLogger | None = None*, ***kwargs*) → None

Create a new KerasModel.

Parameters

- **`model`** (*`tf.keras.Model`*) – the Keras model implementing the calculation
- **`loss`** (*`dc.models.losses.Loss or function`*) – a Loss or function defining how to compute the training loss for each batch, as described above
- **`output_types`** (*`list of strings`*) – the type of each output from the model, as described above
- **`batch_size`** (*`int`*) – default batch size for training and evaluating
- **`model_dir`** (*`str`*) – the directory on disk where the model will be stored. If this is None, a temporary directory is created.
- **`learning_rate`** (*`float or LearningRateSchedule`*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.
- **`optimizer`** (*`Optimizer`*) – the optimizer to use for fitting. If this is specified, learning_rate is ignored.
- **`tensorboard`** (*`bool`*) – whether to log progress to TensorBoard during training
- **`wandb`** (*`bool`*) – whether to log progress to Weights & Biases during training (deprecated)
- **`log_frequency`** (*`int`*) – The frequency at which to log data. Data is logged using *logging* by default. If *tensorboard* is set, data is also logged to TensorBoard. If *wandb* is set, data is also logged to Weights & Biases. Logging happens at global steps. Roughly, a global step corresponds to one batch of training. If you'd like a printout every 10 batch steps, you'd set *log_frequency=10* for example.
- **`wandb_logger`** (*`WandbLogger`*) – the Weights & Biases logger object used to log data and metrics

`fit`(*dataset: Dataset*, *nb_epoch: int = 10*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *deterministic: bool = False*, *restore: bool = False*, *variables: List[Variable] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on a dataset.

Parameters

- **`dataset`** (`Dataset`) – the Dataset to train on
- **`nb_epoch`** (*`int`*) – the number of epochs to train for
- **`max_checkpoints_to_keep`** (*`int`*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **`checkpoint_interval`** (*`int`*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **`deterministic`** (*`bool`*) – if True, the samples are processed in order. If False, a different random order is used for each epoch.
- **`restore`** (*`bool`*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **`variables`** (*`list of tf.Variable`*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **`loss`** (*`function`*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **`callbacks`** (*`function or list of functions`*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **`all_losses`** (*`Optional[List[float]], optional (default None)`*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Returns
The average loss over the most recent checkpoint interval

Return type
float

`fit_generator`(*generator: Iterable[Tuple[Any, Any, Any]]*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *restore: bool = False*, *variables: List[Variable] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on data from a generator.

Parameters

- **`generator`** (*`generator`*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **`max_checkpoints_to_keep`** (*`int`*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **`checkpoint_interval`** (*`int`*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **`restore`** (*`bool`*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **`variables`** (*`list of tf.Variable`*) – the variables to train. If None (the default), all trainable variables in the model are used.

- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step, **kwargs) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Returns
The average loss over the most recent checkpoint interval

Return type
float

fit_on_batch(*X: Sequence*, *y: Sequence*, *w: Sequence*, *variables: List[Variable] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *checkpoint: bool = True*, *max_checkpoints_to_keep: int = 5*) → float

Perform a single step of training.

Parameters

- **X** (*ndarray*) – the inputs for the batch
- **y** (*ndarray*) – the labels for the batch
- **w** (*ndarray*) – the weights for the batch
- **variables** (*list of tf.Variable*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **checkpoint** (*bool*) – if true, save a checkpoint after performing the training step
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.

Returns
the loss on the batch

Return type
float

predict_on_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[Transformer] = []*, *outputs: Tensor | Sequence[Tensor] | None = None*, *output_types: str | Sequence[str] | None = None*) → ndarray | Sequence[ndarray]

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.

- **outputs** (*Tensor or list of Tensors*) – The outputs to return. If this is None, the model's standard prediction outputs will be returned. Alternatively one or more Tensors within the model may be specified, in which case the output of those Tensors will be returned. If outputs is specified, output_types must be None.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.

Returns
a NumPy array of the model produces a single output, or a list of arrays if it produces multiple outputs

Return type
OneOrMany[np.ndarray]

predict_on_batch(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *transformers: List[Transformer] = []*, *outputs: Tensor | Sequence[Tensor] | None = None*) → ndarray | Sequence[ndarray]

Generates predictions for input samples, processing samples in a batch.

Parameters

- **X** (*ndarray*) – the input data, as a Numpy array.
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **outputs** (*Tensor or list of Tensors*) – The outputs to return. If this is None, the model's standard prediction outputs will be returned. Alternatively one or more Tensors within the model may be specified, in which case the output of those Tensors will be returned.

Returns
a NumPy array of the model produces a single output, or a list of arrays if it produces multiple outputs

Return type
OneOrMany[np.ndarray]

predict_uncertainty_on_batch(*X: Sequence*, *masks: int = 50*) → Tuple[ndarray, ndarray] | Sequence[Tuple[ndarray, ndarray]]

Predict the model's outputs, along with the uncertainty in each one.

The uncertainty is computed as described in https://arxiv.org/abs/1703.04977. It involves repeating the prediction many times with different dropout masks. The prediction is computed as the average over all the predictions. The uncertainty includes both the variation among the predicted values (epistemic uncertainty) and the model's own estimates for how well it fits the data (aleatoric uncertainty). Not all models support uncertainty prediction.

Parameters

- **X** (*ndarray*) – the input data, as a Numpy array.
- **masks** (*int*) – the number of dropout masks to average over

Returns

- *OneOrMany[Tuple[y_pred, y_std]]*
- **y_pred** (*np.ndarray*) – predicted value of the output

- **y_std** (*np.ndarray*) – standard deviation of the corresponding element of y_pred

`predict`(*dataset:* Dataset, *transformers: List[*Transformer*] = []*, *outputs: Tensor | Sequence[Tensor] | None = None*, *output_types: List[str] | None = None*) → ndarray | Sequence[ndarray]

Uses self to make predictions on provided Dataset object.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to make prediction on
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **outputs** (*Tensor or list of Tensors*) – The outputs to return. If this is None, the model's standard prediction outputs will be returned. Alternatively one or more Tensors within the model may be specified, in which case the output of those Tensors will be returned.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

`predict_embedding`(*dataset:* Dataset) → ndarray | Sequence[ndarray]

Predicts embeddings created by underlying model if any exist. An embedding must be specified to have *output_type* of *'embedding'* in the model definition.

Parameters
dataset (*dc.data.Dataset*) – Dataset to make prediction on

Returns

- *a NumPy array of the embeddings model produces, or a list*
- *of arrays if it produces multiple embeddings*

`predict_uncertainty`(*dataset:* Dataset, *masks: int = 50*) → Tuple[ndarray, ndarray] | Sequence[Tuple[ndarray, ndarray]]

Predict the model's outputs, along with the uncertainty in each one.

The uncertainty is computed as described in https://arxiv.org/abs/1703.04977. It involves repeating the prediction many times with different dropout masks. The prediction is computed as the average over all the predictions. The uncertainty includes both the variation among the predicted values (epistemic uncertainty) and the model's own estimates for how well it fits the data (aleatoric uncertainty). Not all models support uncertainty prediction.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to make prediction on
- **masks** (*int*) – the number of dropout masks to average over

Returns

- *for each output, a tuple (y_pred, y_std) where y_pred is the predicted*
- *value of the output, and each element of y_std estimates the standard*
- *deviation of the corresponding element of y_pred*

evaluate_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *metrics: List[*Metric*]*, *transformers: List[*Transformer*] = []*, *per_task_metrics: bool = False*)

Evaluate the performance of this model on the data produced by a generator.

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **metric** (*list of* deepchem.metrics.Metric) – Evaluation metric
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **per_task_metrics** (*bool*) – If True, return per-task scores.

Returns
Maps tasks to scores under metric.

Return type
dict

compute_saliency(*X: ndarray*) → ndarray | Sequence[ndarray]

Compute the saliency map for an input sample.

This computes the Jacobian matrix with the derivative of each output element with respect to each input element. More precisely,

- **If this model has a single output, it returns a matrix of shape**
  (output_shape, input_shape) with the derivatives.
- **If this model has multiple outputs, it returns a list of matrices, one**
  for each output.

This method cannot be used on models that take multiple inputs.

Parameters
X (*ndarray*) – the input data for a single sample

Return type
the Jacobian matrix, or a list of matrices

default_generator(*dataset:* Dataset, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

`save_checkpoint`(*max_checkpoints_to_keep: int = 5*, *model_dir: str | None = None*) → None

Save a checkpoint to disk.

Usually you do not need to call this method, since fit() saves checkpoints automatically. If you have disabled automatic checkpointing during fitting, this can be called to manually write checkpoints.

Parameters

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **model_dir** (*str, default None*) – Model directory to save checkpoint to. If None, revert to self.model_dir

`get_checkpoints`(*model_dir: str | None = None*)

Get a list of all available checkpoint files.

Parameters

model_dir (*str, default None*) – Directory to get list of checkpoints from. Reverts to self.model_dir if None

`restore`(*checkpoint: str | None = None*, *model_dir: str | None = None*) → None

Reload the values of all variables from a checkpoint file.

Parameters

- **checkpoint** (*str*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints.
- **model_dir** (*str, default None*) – Directory to restore checkpoint from. If None, use self.model_dir.

`get_global_step`() → int

Get the number of steps of fitting that have been performed.

`load_from_pretrained`(*source_model: KerasModel*, *assignment_map: Dict[Any, Any] | None = None*, *value_map: Dict[Any, Any] | None = None*, *checkpoint: str | None = None*, *model_dir: str | None = None*, *include_top: bool = True*, *inputs: Sequence[Any] | None = None*, ***kwargs*) → None

Copies variable values from a pretrained model. *source_model* can either be a pretrained model or a model with the same architecture. *value_map* is a variable-value dictionary. If no *value_map* is provided, the variable values are restored to the *source_model* from a checkpoint and a default *value_map* is created. *assignment_map* is a dictionary mapping variables from the *source_model* to the current model. If no *assignment_map* is provided, one is made from scratch and assumes the model is composed of several different layers, with the final one being a dense layer. *include_top* is used to control whether or not the final dense layer is used. The default assignment map is useful in cases where the type of task is different (classification vs regression) and/or number of tasks in the setting.

Parameters

- **source_model** (*dc.KerasModel, required*) – source_model can either be the pre-trained model or a dc.KerasModel with the same architecture as the pretrained model. It is used to restore from a checkpoint, if value_map is None and to create a default assignment map if assignment_map is None

- **assignment_map** (*Dict, default None*) – Dictionary mapping the source_model variables and current model variables
- **value_map** (*Dict, default None*) – Dictionary containing source_model trainable variables mapped to numpy arrays. If value_map is None, the values are restored and a default variable map is created using the restored values
- **checkpoint** (*str, default None*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints
- **model_dir** (*str, default None*) – Restore model from custom model directory if needed
- **include_top** (*bool, default True*) – if True, copies the weights and bias associated with the final dense layer. Used only when assignment map is None
- **inputs** (*List, input tensors for model*) – if not None, then the weights are built for both the source and self. This option is useful only for models that are built by subclassing tf.keras.Model, and not using the functional API by tf.keras

3.17.2 TensorflowMultitaskIRVClassifier

class **TensorflowMultitaskIRVClassifier**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

Initialize MultitaskIRVClassifier

Parameters

- **n_tasks** (*int*) – Number of tasks
- **K** (*int*) – Number of nearest neighbours used in classification
- **penalty** (*float*) – Amount of penalty (l2 or l1 applied)

3.17.3 RobustMultitaskClassifier

class **RobustMultitaskClassifier**(*n_tasks, n_features, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, n_classes=2, bypass_layer_sizes=[100], bypass_weight_init_stddevs=[0.02], bypass_bias_init_consts=[1.0], bypass_dropouts=[0.5], **kwargs*)

Implements a neural network for robust multitasking.

The key idea of this model is to have bypass layers that feed directly from features to task output. This might provide some flexibility toroute around challenges in multitasking with destructive interference.

References

This technique was introduced in [1]_

`__init__`(*n_tasks, n_features, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, n_classes=2, bypass_layer_sizes=[100], bypass_weight_init_stddevs=[0.02], bypass_bias_init_consts=[1.0], bypass_dropouts=[0.5], **kwargs*)

Create a RobustMultitaskClassifier.

Parameters

- **n_tasks** (*int*) – number of tasks
- **n_features** (*int*) – number of features
- **layer_sizes** (*list*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or loat*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float*) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str*) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float*) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (*list or object*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **n_classes** (*int*) – the number of classes
- **bypass_layer_sizes** (*list*) – the size of each dense layer in the bypass network. The length of this list determines the number of bypass layers.
- **bypass_weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of bypass layers. same requirements as weight_init_stddevs
- **bypass_bias_init_consts** (*list or float*) – the value to initialize the biases in bypass layers same requirements as bias_init_consts
- **bypass_dropouts** (*list or float*) – the dropout probablity to use for bypass layers. same requirements as dropouts

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.4 RobustMultitaskRegressor

class RobustMultitaskRegressor(*n_tasks, n_features, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, bypass_layer_sizes=[100], bypass_weight_init_stddevs=[0.02], bypass_bias_init_consts=[1.0], bypass_dropouts=[0.5], **kwargs*)

Implements a neural network for robust multitasking.

The key idea of this model is to have bypass layers that feed directly from features to task output. This might provide some flexibility to route around challenges in multitasking with destructive interference.

References

__init__(*n_tasks, n_features, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, bypass_layer_sizes=[100], bypass_weight_init_stddevs=[0.02], bypass_bias_init_consts=[1.0], bypass_dropouts=[0.5], **kwargs*)

Create a RobustMultitaskRegressor.

Parameters

- **n_tasks** (*int*) – number of tasks
- **n_features** (*int*) – number of features
- **layer_sizes** (*list*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or loat*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float*) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str*) – the type of penalty to use for weight decay, either 'l1' or 'l2'

- **dropouts** (*list or float*) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (*list or object*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bypass_layer_sizes** (*list*) – the size of each dense layer in the bypass network. The length of this list determines the number of bypass layers.
- **bypass_weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of bypass layers. same requirements as weight_init_stddevs
- **bypass_bias_init_consts** (*list or float*) – the value to initialize the biases in bypass layers same requirements as bias_init_consts
- **bypass_dropouts** (*list or float*) – the dropout probablity to use for bypass layers. same requirements as dropouts

default_generator(*dataset: Dataset, epochs: int = 1, mode: str = 'fit', deterministic: bool = True, pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.5 ProgressiveMultitaskClassifier

class ProgressiveMultitaskClassifier(*n_tasks, n_features, alpha_init_stddevs=0.02, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, **kwargs*)

Implements a progressive multitask neural network for classification.

Progressive Networks: https://arxiv.org/pdf/1606.04671v3.pdf

Progressive networks allow for multitask learning where each task gets a new column of weights. As a result, there is no exponential forgetting where previous tasks are ignored.

__init__(*n_tasks, n_features, alpha_init_stddevs=0.02, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, **kwargs*)

Creates a progressive network.

Only listing parameters specific to progressive networks here.

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_features** (*int*) – Number of input features
- **alpha_init_stddevs** (*list*) – List of standard-deviations for alpha in adapter layers.
- **layer_sizes** (*list*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or float*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float*) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str*) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float*) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (*list or object*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

3.17.6 ProgressiveMultitaskRegressor

class **ProgressiveMultitaskRegressor**(*n_tasks, n_features, alpha_init_stddevs=0.02, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, n_outputs=1, **kwargs*)

Implements a progressive multitask neural network for regression.

Progressive networks allow for multitask learning where each task gets a new column of weights. As a result, there is no exponential forgetting where previous tasks are ignored.

References

See [1]_ for a full description of the progressive architecture

`__init__`(*n_tasks, n_features, alpha_init_stddevs=0.02, layer_sizes=[1000], weight_init_stddevs=0.02, bias_init_consts=1.0, weight_decay_penalty=0.0, weight_decay_penalty_type='l2', dropouts=0.5, activation_fns=<function relu>, n_outputs=1, **kwargs*)

Creates a progressive network.

Only listing parameters specific to progressive networks here.

Parameters

- **`n_tasks`** (*`int`*) – Number of tasks
- **`n_features`** (*`int`*) – Number of input features
- **`alpha_init_stddevs`** (*`list`*) – List of standard-deviations for alpha in adapter layers.
- **`layer_sizes`** (*`list`*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **`weight_init_stddevs`** (*`list or float`*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **`bias_init_consts`** (*`list or float`*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **`weight_decay_penalty`** (*`float`*) – the magnitude of the weight decay penalty to use
- **`weight_decay_penalty_type`** (*`str`*) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **`dropouts`** (*`list or float`*) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **`activation_fns`** (*`list or object`*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

`add_adapter`(*all_layers*, *task*, *layer_num*)

Add an adapter connection for given task/layer combo

`fit`(*dataset*, *nb_epoch=10*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *deterministic=False*, *restore=False*, ***kwargs*)

Train this model on a dataset.

Parameters

- **`dataset`** (`Dataset`) – the Dataset to train on
- **`nb_epoch`** (*`int`*) – the number of epochs to train for
- **`max_checkpoints_to_keep`** (*`int`*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **`checkpoint_interval`** (*`int`*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.

- **deterministic** (*bool*) – if True, the samples are processed in order. If False, a different random order is used for each epoch.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of tf.Variable*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Returns
The average loss over the most recent checkpoint interval

Return type
float

fit_task(*dataset*, *task*, *nb_epoch=10*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *deterministic=False*, *restore=False*, ***kwargs*)

Fit one task.

3.17.7 WeaveModel

class WeaveModel(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn: ~typing.Callable | str | None = <function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = <function relu>, batch_normalize: bool = True, batch_normalize_kwargs: ~typing.Dict = {'fused': False, 'renorm': True}, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100, **kwargs*)

Implements Google-style Weave Graph Convolutions

This model implements the Weave style graph convolutions from [1]_.

The biggest difference between WeaveModel style convolutions and GraphConvModel style convolutions is that Weave convolutions model bond features explicitly. This has the side effect that it needs to construct a NxN matrix explicitly to model bond interactions. This may cause scaling issues, but may possibly allow for better modeling of subtle bond effects.

Note that [1]_ introduces a whole variety of different architectures for Weave models. The default settings in this class correspond to the W2N2 variant from [1]_ which is the most commonly used variant..

Examples

Here's an example of how to fit a *WeaveModel* on a tiny sample dataset.

```
>>> import numpy as np
>>> import deepchem as dc
>>> featurizer = dc.feat.WeaveFeaturizer()
>>> X = featurizer(["C", "CC"])
>>> y = np.array([1, 0])
>>> dataset = dc.data.NumpyDataset(X, y)
>>> model = dc.models.WeaveModel(n_tasks=1, n_weave=2, fully_connected_layer_
↪sizes=[2000, 1000], mode="classification")
>>> loss = model.fit(dataset)
```

Note: In general, the use of batch normalization can cause issues with NaNs. If you're having trouble with NaNs while using this model, consider setting *batch_normalize_kwargs={"trainable": False}* or turning off batch normalization entirely with *batch_normalize=False*.

References

__init__(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn: ~typing.Callable | str | None = <function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = <function relu>, batch_normalize: bool = True, batch_normalize_kwargs: ~typing.Dict = {'fused': False, 'renorm': True}, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100, **kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_atom_feat** (*int, optional (default 75)*) – Number of features per atom. Note this is 75 by default and should be 78 if chirality is used by *WeaveFeaturizer*.
- **n_pair_feat** (*int, optional (default 14)*) – Number of features per pair of atoms.
- **n_hidden** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_graph_feat** (*int, optional (default 128)*) – Number of output features for each molecule(graph)
- **n_weave** (*int, optional (default 2)*) – The number of weave layers in this model.
- **fully_connected_layer_sizes** (list (default *[2000, 100]*)) – The size of each dense layer in the network. The length of this list determines the number of layers.
- **conv_weight_init_stddevs** (*list or float (default 0.03)*) – The standard deviation of the distribution to use for weight initialization of each convolutional layer. The length of this lisst should equal *n_weave*. Alternatively, this may be a single value instead of a list, in which case the same value is used for each layer.

- **weight_init_stddevs** (*list or float (default 0.01)*) – The standard deviation of the distribution to use for weight initialization of each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or float (default 0.0)*) – The value to initialize the biases in each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float (default 0.0)*) – The magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str (default "l2")*) – The type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float (default 0.25)*) – The dropout probablity to use for each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **final_conv_activation_fn** (Optional[ActivationFn] (default *tf.nn.tanh*)) – The Tensorflow activation funcntion to apply to the final convolution at the end of the weave convolutions. If *None*, then no activate is applied (hence linear).
- **activation_fns** (list or object (default *tf.nn.relu*)) – The Tensorflow activation function to apply to each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **batch_normalize** (*bool, optional (default True)*) – If this is turned on, apply batch normalization before applying activation functions on convolutional and fully connected layers.
- **batch_normalize_kwargs** (Dict, optional (default *{"renorm"=True, "fused": False}*)) – Batch normalization is a complex layer which has many potential argumentswhich change behavior. This layer accepts user-defined parameters which are passed to all *BatchNormalization* layers in *WeaveModel*, *WeaveLayer*, and *WeaveGather*.
- **gaussian_expand** (*boolean, optional (default True)*) – Whether to expand each dimension of atomic features by gaussian histogram
- **compress_post_gaussian_expansion** (*bool, optional (default False)*) – If True, compress the results of the Gaussian expansion back to the original dimensions of the input.
- **mode** (*str (default "classification")*) – Either "classification" or "regression" for type of model.
- **n_classes** (*int (default 2)*) – Number of classes to predict (only used in classification mode)
- **batch_size** (*int (default 100)*) – Batch size used by this model for training.

compute_features_on_batch(*X_b*)

Compute tensors that will be input into the model from featurized representation.

The featurized input to *WeaveModel* is instances of *WeaveMol* created by *WeaveFeaturizer*. This method converts input *WeaveMol* objects into tensors used by the Keras implementation to compute *WeaveModel* outputs.

Parameters
`X_b` (*np.ndarray*) – A numpy array with dtype=object where elements are *WeaveMol* objects.

Returns

- **atom_feat** (*np.ndarray*) – Of shape *(N_atoms, N_atom_feat)*.
- **pair_feat** (*np.ndarray*) – Of shape *(N_pairs, N_pair_feat)*. Note that *N_pairs* will depend on the number of pairs being considered. If *max_pair_distance* is *None*, then this will be *N_atoms**2*. Else it will be the number of pairs within the specifed graph distance.
- **pair_split** (*np.ndarray*) – Of shape *(N_pairs,)*. The i-th entry in this array will tell you the originating atom for this pair (the "source"). Note that pairs are symmetric so for a pair *(a, b)*, both *a* and *b* will separately be sources at different points in this array.
- **atom_split** (*np.ndarray*) – Of shape *(N_atoms,)*. The i-th entry in this array will be the molecule with the i-th atom belongs to.
- **atom_to_pair** (*np.ndarray*) – Of shape *(N_pairs, 2)*. The i-th row in this array will be the array *[a, b]* if *(a, b)* is a pair to be considered. (Note by symmetry, this implies some other row will contain *[b, a]*.

`default_generator`(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Convert a dataset into the tensors needed for learning.

Parameters

- `dataset` (*dc.data.Dataset*) – Dataset to convert
- `epochs` (`int, optional (Default 1)`) – Number of times to walk over *dataset*
- `mode` (`str, optional (Default 'fit')`) – Ignored in this implementation.
- `deterministic` (`bool, optional (Default True)`) – Whether the dataset should be walked in a deterministic fashion
- `pad_batches` (`bool, optional (Default True)`) – If true, each returned batch will have size *self.batch_size*.

Return type
Iterator which walks over the batches

3.17.8 DTNNModel

class DTNNModel(*n_tasks*, *n_embedding=30*, *n_hidden=100*, *n_distance=100*, *distance_min=-1*, *distance_max=18*, *output_activation=True*, *mode='regression'*, *dropout=0.0*, ***kwargs*)

Deep Tensor Neural Networks

This class implements deep tensor neural networks as first defined in **[1]_**

References

`__init__`(*n_tasks*, *n_embedding=30*, *n_hidden=100*, *n_distance=100*, *distance_min=-1*, *distance_max=18*, *output_activation=True*, *mode='regression'*, *dropout=0.0*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_embedding** (*int, optional*) – Number of features per atom.
- **n_hidden** (*int, optional*) – Number of features for each molecule after DTNNStep
- **n_distance** (*int, optional*) – granularity of distance matrix step size will be (distance_max-distance_min)/n_distance
- **distance_min** (*float, optional*) – minimum distance of atom pairs, default = -1 Angstorm
- **distance_max** (*float, optional*) – maximum distance of atom pairs, default = 18 Angstorm
- **mode** (*str*) – Only "regression" is currently supported.
- **dropout** (*float*) – the dropout probablity to use.

`compute_features_on_batch`(*X_b*)

Computes the values for different Feature Layers on given batch

A tf.py_func wrapper is written around this when creating the input_fn for tf.Estimator

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.9 DAGModel

`class DAGModel`(*n_tasks*, *max_atoms=50*, *n_atom_feat=75*, *n_graph_feat=30*, *n_outputs=30*, *layer_sizes=[100]*, *layer_sizes_gather=[100]*, *dropout=None*, *mode='classification'*, *n_classes=2*, *uncertainty=False*, *batch_size=100*, ***kwargs*)

Directed Acyclic Graph models for molecular property prediction.

This model is based on the following paper:

Lusci, Alessandro, Gianluca Pollastri, and Pierre Baldi. "Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules." Journal of chemical information and modeling 53.7 (2013): 1563-1575.

The basic idea for this paper is that a molecule is usually viewed as an undirected graph. However, you can convert it to a series of directed graphs. The idea is that for each atom, you make a DAG using that atom as the vertex of the DAG and edges pointing "inwards" to it. This transformation is implemented in *dc.trans.transformers.DAGTransformer.UG_to_DAG*.

This model accepts ConvMols as input, just as GraphConvModel does, but these ConvMol objects must be transformed by dc.trans.DAGTransformer.

As a note, performance of this model can be a little sensitive to initialization. It might be worth training a few different instantiations to get a stable set of parameters.

`__init__`(*n_tasks*, *max_atoms=50*, *n_atom_feat=75*, *n_graph_feat=30*, *n_outputs=30*, *layer_sizes=[100]*, *layer_sizes_gather=[100]*, *dropout=None*, *mode='classification'*, *n_classes=2*, *uncertainty=False*, *batch_size=100*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.
- **max_atoms** (*int, optional*) – Maximum number of atoms in a molecule, should be defined based on dataset.
- **n_atom_feat** (*int, optional*) – Number of features per atom.
- **n_graph_feat** (*int, optional*) – Number of features for atom in the graph.
- **n_outputs** (*int, optional*) – Number of features for each molecule.
- **layer_sizes** (*list of int, optional*) – List of hidden layer size(s) in the propagation step: length of this list represents the number of hidden layers, and each element is the width of corresponding hidden layer.
- **layer_sizes_gather** (*list of int, optional*) – List of hidden layer size(s) in the gather step.
- **dropout** (*None or float, optional*) – Dropout probability, applied after each propagation step and gather step.
- **mode** (*str, optional*) – Either "classification" or "regression" for type of model.
- **n_classes** (*int*) – the number of classes to predict (only used in classification mode)
- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Convert a dataset into the tensors needed for learning

3.17.10 GraphConvModel

class GraphConvModel(*n_tasks: int*, *graph_conv_layers: List[int] = [64, 64]*, *dense_layer_size: int = 128*, *dropout: float = 0.0*, *mode: str = 'classification'*, *number_atom_features: int = 75*, *n_classes: int = 2*, *batch_size: int = 100*, *batch_normalize: bool = True*, *uncertainty: bool = False*, ***kwargs*)

Graph Convolutional Models.

This class implements the graph convolutional model from the following paper [1]_. These graph convolutions start with a per-atom set of descriptors for each atom in a molecule, then combine and recombine these descriptors over convolutional layers. following [1]_.

References

__init__(*n_tasks: int*, *graph_conv_layers: List[int] = [64, 64]*, *dense_layer_size: int = 128*, *dropout: float = 0.0*, *mode: str = 'classification'*, *number_atom_features: int = 75*, *n_classes: int = 2*, *batch_size: int = 100*, *batch_normalize: bool = True*, *uncertainty: bool = False*, ***kwargs*)

The wrapper class for graph convolutions.

Note that since the underlying _GraphConvKerasModel class is specified using imperative subclassing style, this model cannout make predictions for arbitrary outputs.

Parameters

- **n_tasks** (*int*) – Number of tasks
- **graph_conv_layers** (*list of int*) – Width of channels for the Graph Convolution Layers
- **dense_layer_size** (*int*) – Width of channels for Atom Level Dense Layer after GraphPool
- **dropout** (*list or float*) – the dropout probablity to use for each layer. The length of this list should equal len(graph_conv_layers)+1 (one value for each convolution layer, and one for the dense layer). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **mode** (*str*) – Either "classification" or "regression"
- **number_atom_features** (*int*) – 75 is the default number of atom features created, but this can vary if various options are passed to the function atom_features in graph_features
- **n_classes** (*int*) – the number of classes to predict (only used in classification mode)
- **batch_normalize** (*True*) – if True, apply batch normalization to model
- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted

default_generator(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (*Dataset*) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset

- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.11 MPNNModel

class MPNNModel(*n_tasks*, *n_atom_feat=70*, *n_pair_feat=8*, *n_hidden=100*, *T=5*, *M=10*, *mode='regression'*, *dropout=0.0*, *n_classes=2*, *uncertainty=False*, *batch_size=100*, ***kwargs*)

Message Passing Neural Network,

Message Passing Neural Networks [1]_ treat graph convolutional operations as an instantiation of a more general message passing schem. Recall that message passing in a graph is when nodes in a graph send each other "messages" and update their internal state as a consequence of these messages.

Ordering structures in this model are built according to [2]_

References

__init__(*n_tasks*, *n_atom_feat=70*, *n_pair_feat=8*, *n_hidden=100*, *T=5*, *M=10*, *mode='regression'*, *dropout=0.0*, *n_classes=2*, *uncertainty=False*, *batch_size=100*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_atom_feat** (*int, optional*) – Number of features per atom.
- **n_pair_feat** (*int, optional*) – Number of features per pair of atoms.
- **n_hidden** (*int, optional*) – Number of units(convolution depths) in corresponding hidden layer
- **n_graph_feat** (*int, optional*) – Number of output features for each molecule(graph)
- **dropout** (*float*) – the dropout probablity to use.
- **n_classes** (*int*) – the number of classes to predict (only used in classification mode)
- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted

default_generator(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (*Dataset*) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset

- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.12 BasicMolGANModel

class BasicMolGANModel(*edges: int = 5*, *vertices: int = 9*, *nodes: int = 5*, *embedding_dim: int = 10*, *dropout_rate: float = 0.0*, ***kwargs*)

Model for de-novo generation of small molecules based on work of Nicola De Cao et al. [1]_. It uses a GAN directly on graph data and a reinforcement learning objective to induce the network to generate molecules with certain chemical properties. Utilizes WGAN infrastructure; uses adjacency matrix and node features as inputs. Inputs need to be one-hot representation.

Examples

```
>>>
>> import deepchem as dc
>> from deepchem.models import BasicMolGANModel as MolGAN
>> from deepchem.models.optimizers import ExponentialDecay
>> from tensorflow import one_hot
>> smiles = ['CCC', 'C1=CC=CC=C1', 'CNC' ]
>> # create featurizer
>> feat = dc.feat.MolGanFeaturizer()
>> # featurize molecules
>> features = feat.featurize(smiles)
>> # Remove empty objects
>> features = list(filter(lambda x: x is not None, features))
>> # create model
>> gan = MolGAN(learning_rate=ExponentialDecay(0.001, 0.9, 5000))
>> dataset = dc.data.NumpyDataset([x.adjacency_matrix for x in features],[x.node_
↪features for x in features])
>> def iterbatches(epochs):
>>     for i in range(epochs):
>>         for batch in dataset.iterbatches(batch_size=gan.batch_size, pad_
↪batches=True):
>>             adjacency_tensor = one_hot(batch[0], gan.edges)
>>             node_tensor = one_hot(batch[1], gan.nodes)
>>             yield {gan.data_inputs[0]: adjacency_tensor, gan.data_inputs[1]:node_
↪tensor}
>> gan.fit_gan(iterbatches(8), generator_steps=0.2, checkpoint_interval=5000)
>> generated_data = gan.predict_gan_generator(1000)
>> # convert graphs to RDKitmolecules
>> nmols = feat.defeaturize(generated_data)
```

(continues on next page)

(continued from previous page)

```
>> print("{} molecules generated".format(len(nmols)))
>> # remove invalid moles
>> nmols = list(filter(lambda x: x is not None, nmols))
>> # currently training is unstable so 0 is a common outcome
>> print ("{} valid molecules".format(len(nmols)))
```

References

`__init__`(*edges: int = 5*, *vertices: int = 9*, *nodes: int = 5*, *embedding_dim: int = 10*, *dropout_rate: float = 0.0*, ***kwargs*)

Initialize the model

Parameters

- **edges** (*int, default 5*) – Number of bond types includes BondType.Zero
- **vertices** (*int, default 9*) – Max number of atoms in adjacency and node features matrices
- **nodes** (*int, default 5*) – Number of atom types in node features matrix
- **embedding_dim** (*int, default 10*) – Size of noise input array
- **dropout_rate** (*float, default = 0.*) – Rate of dropout used across whole model
- **name** (*str, default ''*) – Name of the model

`get_noise_input_shape`() → Tuple[int]

Return shape of the noise input used in generator

Returns
Shape of the noise input

Return type
Tuple

`get_data_input_shapes`() → List

Return input shape of the discriminator

Returns
List of shapes used as an input for distriminator.

Return type
List

`create_generator`() → Model

Create generator model. Take noise data as an input and processes it through number of dense and dropout layers. Then data is converted into two forms one used for training and other for generation of compounds. The model has two outputs:

1. edges
2. nodes

The format differs depending on intended use (training or sample generation). For sample generation use flag, sample_generation=True while calling generator i.e. gan.generators[0](noise_input, training=False, sample_generation=True). For training the model, set *sample_generation=False*

create_discriminator() → Model

Create discriminator model based on MolGAN layers. Takes two inputs:

1. adjacency tensor, containing bond information
2. nodes tensor, containing atom information

The input vectors need to be in one-hot encoding format. Use MolGAN featurizer for that purpose. It will be simplified in the future release.

predict_gan_generator(*batch_size: int = 1*, *noise_input: List | None = None*, *conditional_inputs: List = []*, *generator_index: int = 0*) → List[GraphMatrix]

Use the GAN to generate a batch of samples.

Parameters

- **batch_size** (*int*) – the number of samples to generate. If either noise_input or conditional_inputs is specified, this argument is ignored since the batch size is then determined by the size of that argument.
- **noise_input** (*array*) – the value to use for the generator's noise input. If None (the default), get_noise_batch() is called to generate a random input, so each call will produce a new set of samples.
- **conditional_inputs** (*list of arrays*) – NOT USED. the values to use for all conditional inputs. This must be specified if the GAN has any conditional inputs.
- **generator_index** (*int*) – NOT USED. the index of the generator (between 0 and n_generators-1) to use for generating the samples.

Returns

Returns a list of GraphMatrix object that can be converted into RDKit molecules using MolGANFeaturizer defeaturize function.

Return type

List[GraphMatrix]

3.17.13 ScScoreModel

class **ScScoreModel**(*n_features*, *layer_sizes=[300, 300, 300]*, *dropouts=0.0*, ***kwargs*)

The SCScore model is a neural network model based on the work of Coley et al. [1]_ that predicts the synthetic complexity score (SCScore) of molecules and correlates it with the expected number of reaction steps required to produce the given target molecule. It is trained on a dataset of over 12 million reactions from the Reaxys database to impose a pairwise inequality constraint enforcing that on average the products of published chemical reactions should be more synthetically complex than their corresponding reactants. The learned metric (SCScore) exhibits highly desirable nonlinear behavior, particularly in recognizing increases in synthetic complexity throughout a number of linear synthetic routes. The SCScore model can accurately predict the synthetic complexity of a variety of molecules, including both drug-like and natural product molecules. SCScore has the potential to be a valuable tool for chemists who are working on drug discovery and other areas of chemistry.

The learned metric (SCScore) exhibits highly desirable nonlinear behavior, particularly in recognizing increases in synthetic complexity throughout a number of linear synthetic routes.

Our model uses hingeloss instead of the shifted relu loss as in the supplementary material [2]_ provided by the author. This could cause differentiation issues with compounds that are "close" to each other in "complexity".

References

`__init__`(*n_features*, *layer_sizes=[300, 300, 300]*, *dropouts=0.0*, ***kwargs*)

Parameters

- **n_features** (*int*) – number of features per molecule
- **layer_sizes** (*list of int*) – size of each hidden layer
- **dropouts** (*int*) – droupout to apply to each hidden layer
- **kwargs** – This takes all kwards as TensorGraph

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.14 SeqToSeq

`class SeqToSeq`(*input_tokens*, *output_tokens*, *max_output_length*, *encoder_layers=4*, *decoder_layers=4*, *embedding_dimension=512*, *dropout=0.0*, *reverse_input=True*, *variational=False*, *annealing_start_step=5000*, *annealing_final_step=10000*, ***kwargs*)

Implements sequence to sequence translation models.

The model is based on the description in Sutskever et al., "Sequence to Sequence Learning with Neural Networks" (https://arxiv.org/abs/1409.3215), although this implementation uses GRUs instead of LSTMs. The goal is to take sequences of tokens as input, and translate each one into a different output sequence. The input and output sequences can both be of variable length, and an output sequence need not have the same length as the input sequence it was generated from. For example, these models were originally developed for use in natural language processing. In that context, the input might be a sequence of English words, and the output might be a sequence of French words. The goal would be to train the model to translate sentences from English to French.

The model consists of two parts called the "encoder" and "decoder". Each one consists of a stack of recurrent layers. The job of the encoder is to transform the input sequence into a single, fixed length vector called the "embedding". That vector contains all relevant information from the input sequence. The decoder then transforms the embedding vector into the output sequence.

These models can be used for various purposes. First and most obviously, they can be used for sequence to sequence translation. In any case where you have sequences of tokens, and you want to translate each one into a different sequence, a SeqToSeq model can be trained to perform the translation.

Another possible use case is transforming variable length sequences into fixed length vectors. Many types of models require their inputs to have a fixed shape, which makes it difficult to use them with variable sized inputs (for example, when the input is a molecule, and different molecules have different numbers of atoms). In that case, you can train a SeqToSeq model as an autoencoder, so that it tries to make the output sequence identical to the input one. That forces the embedding vector to contain all information from the original sequence. You can then use the encoder for transforming sequences into fixed length embedding vectors, suitable to use as inputs to other types of models.

Another use case is to train the decoder for use as a generative model. Here again you begin by training the SeqToSeq model as an autoencoder. Once training is complete, you can supply arbitrary embedding vectors, and transform each one into an output sequence. When used in this way, you typically train it as a variational autoencoder. This adds random noise to the encoder, and also adds a constraint term to the loss that forces the embedding vector to have a unit Gaussian distribution. You can then pick random vectors from a Gaussian distribution, and the output sequences should follow the same distribution as the training data.

When training as a variational autoencoder, it is best to use KL cost annealing, as described in https://arxiv.org/abs/1511.06349. The constraint term in the loss is initially set to 0, so the optimizer just tries to minimize the reconstruction loss. Once it has made reasonable progress toward that, the constraint term can be gradually turned back on. The range of steps over which this happens is configurable.

`__init__`(*input_tokens*, *output_tokens*, *max_output_length*, *encoder_layers=4*, *decoder_layers=4*, *embedding_dimension=512*, *dropout=0.0*, *reverse_input=True*, *variational=False*, *annealing_start_step=5000*, *annealing_final_step=10000*, ***kwargs*)

Construct a SeqToSeq model.

In addition to the following arguments, this class also accepts all the keyword arguments from TensorGraph.

Parameters

- **input_tokens** (*list*) – a list of all tokens that may appear in input sequences
- **output_tokens** (*list*) – a list of all tokens that may appear in output sequences
- **max_output_length** (*int*) – the maximum length of output sequence that may be generated
- **encoder_layers** (*int*) – the number of recurrent layers in the encoder
- **decoder_layers** (*int*) – the number of recurrent layers in the decoder
- **embedding_dimension** (*int*) – the width of the embedding vector. This also is the width of all recurrent layers.
- **dropout** (*float*) – the dropout probability to use during training
- **reverse_input** (*bool*) – if True, reverse the order of input sequences before sending them into the encoder. This can improve performance when working with long sequences.
- **variational** (*bool*) – if True, train the model as a variational autoencoder. This adds random noise to the encoder, and also constrains the embedding to follow a unit Gaussian distribution.
- **annealing_start_step** (*int*) – the step (that is, batch) at which to begin turning on the constraint term for KL cost annealing
- **annealing_final_step** (*int*) – the step (that is, batch) at which to finish turning on the constraint term for KL cost annealing

`fit_sequences`(*sequences*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *restore=False*)

Train this model on a set of sequences

Parameters

- **sequences** (*iterable*) – the training samples to fit to. Each sample should be represented as a tuple of the form (input_sequence, output_sequence).
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

`predict_from_sequences`(*sequences*, *beam_width=5*)

Given a set of input sequences, predict the output sequences.

The prediction is done using a beam search with length normalization.

Parameters

- **sequences** (*iterable*) – the input sequences to generate a prediction for
- **beam_width** (*int*) – the beam width to use for searching. Set to 1 to use a simple greedy search.

`predict_from_embeddings`(*embeddings*, *beam_width=5*)

Given a set of embedding vectors, predict the output sequences.

The prediction is done using a beam search with length normalization.

Parameters

- **embeddings** (*iterable*) – the embedding vectors to generate predictions for
- **beam_width** (*int*) – the beam width to use for searching. Set to 1 to use a simple greedy search.

`predict_embeddings`(*sequences*)

Given a set of input sequences, compute the embedding vectors.

Parameters

sequences (*iterable*) – the input sequences to generate an embedding vector for

3.17.15 GAN

`class GAN`(*n_generators=1*, *n_discriminators=1*, ***kwargs*)

Implements Generative Adversarial Networks.

A Generative Adversarial Network (GAN) is a type of generative model. It consists of two parts called the “generator” and the “discriminator”. The generator takes random noise as input and transforms it into an output that (hopefully) resembles the training data. The discriminator takes a set of samples as input and tries to distinguish the real training samples from the ones created by the generator. Both of them are trained together. The discriminator tries to get better and better at telling real from false data, while the generator tries to get better and better at fooling the discriminator.

In many cases there also are additional inputs to the generator and discriminator. In that case it is known as a Conditional GAN (CGAN), since it learns a distribution that is conditional on the values of those inputs. They are referred to as “conditional inputs”.

Many variations on this idea have been proposed, and new varieties of GANs are constantly being proposed. This class tries to make it very easy to implement straightforward GANs of the most conventional types. At the same time, it tries to be flexible enough that it can be used to implement many (but certainly not all) variations on the concept.

To define a GAN, you must create a subclass that provides implementations of the following methods:

get_noise_input_shape() get_data_input_shapes() create_generator() create_discriminator()

If you want your GAN to have any conditional inputs you must also implement:

get_conditional_input_shapes()

The following methods have default implementations that are suitable for most conventional GANs. You can override them if you want to customize their behavior:

create_generator_loss() create_discriminator_loss() get_noise_batch()

This class allows a GAN to have multiple generators and discriminators, a model known as MIX+GAN. It is described in Arora et al., "Generalization and Equilibrium in Generative Adversarial Nets (GANs)" (https://arxiv.org/abs/1703.00573). This can lead to better models, and is especially useful for reducing mode collapse, since different generators can learn different parts of the distribution. To use this technique, simply specify the number of generators and discriminators when calling the constructor. You can then tell predict_gan_generator() which generator to use for predicting samples.

`__init__`(*n_generators=1*, *n_discriminators=1*, ***kwargs*)

Construct a GAN.

In addition to the parameters listed below, this class accepts all the keyword arguments from KerasModel.

Parameters

- **n_generators** (*int*) – the number of generators to include
- **n_discriminators** (*int*) – the number of discriminators to include

`get_noise_input_shape()`

Get the shape of the generator's noise input layer.

Subclasses must override this to return a tuple giving the shape of the noise input. The actual Input layer will be created automatically. The dimension corresponding to the batch size should be omitted.

`get_data_input_shapes()`

Get the shapes of the inputs for training data.

Subclasses must override this to return a list of tuples, each giving the shape of one of the inputs. The actual Input layers will be created automatically. This list of shapes must also match the shapes of the generator's outputs. The dimension corresponding to the batch size should be omitted.

`get_conditional_input_shapes()`

Get the shapes of any conditional inputs.

Subclasses may override this to return a list of tuples, each giving the shape of one of the conditional inputs. The actual Input layers will be created automatically. The dimension corresponding to the batch size should be omitted.

The default implementation returns an empty list, meaning there are no conditional inputs.

`get_noise_batch`(*batch_size*)

Get a batch of random noise to pass to the generator.

This should return a NumPy array whose shape matches the one returned by get_noise_input_shape(). The default implementation returns normally distributed values. Subclasses can override this to implement a different distribution.

`create_generator()`

Create and return a generator.

Subclasses must override this to construct the generator. The returned value should be a tf.keras.Model whose inputs are a batch of noise, followed by any conditional inputs. The number and shapes of its outputs must match the return value from get_data_input_shapes(), since generated data must have the same form as training data.

`create_discriminator()`

Create and return a discriminator.

Subclasses must override this to construct the discriminator. The returned value should be a tf.keras.Model whose inputs are all data inputs, followed by any conditional inputs. Its output should be a one dimensional tensor containing the probability of each sample being a training sample.

`create_generator_loss`(*discrim_output*)

Create the loss function for the generator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters

discrim_output (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Return type

A Tensor equal to the loss function to use for optimizing the generator.

`create_discriminator_loss`(*discrim_output_train*, *discrim_output_gen*)

Create the loss function for the discriminator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters

- **discrim_output_train** (*Tensor*) – the output from the discriminator on a batch of training data. This is its estimate of the probability that each sample is training data.
- **discrim_output_gen** (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Return type

A Tensor equal to the loss function to use for optimizing the discriminator.

`fit_gan`(*batches*, *generator_steps=1.0*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *restore=False*)

Train this model on data.

Parameters

- **batches** (*iterable*) – batches of data to train the discriminator on, each represented as a dict that maps Inputs to values. It should specify values for all members of data_inputs and conditional_inputs.
- **generator_steps** (*float*) – the number of training steps to perform for the generator for each batch. This can be used to adjust the ratio of training steps for the generator and discriminator. For example, 2.0 will perform two training steps for every batch, while 0.5 will only perform one training step for every two batches.
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.

- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in batches. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint before training it.

predict_gan_generator(*batch_size=1*, *noise_input=None*, *conditional_inputs=[]*, *generator_index=0*)

Use the GAN to generate a batch of samples.

Parameters

- **batch_size** (*int*) – the number of samples to generate. If either noise_input or conditional_inputs is specified, this argument is ignored since the batch size is then determined by the size of that argument.
- **noise_input** (*array*) – the value to use for the generator's noise input. If None (the default), get_noise_batch() is called to generate a random input, so each call will produce a new set of samples.
- **conditional_inputs** (*list of arrays*) – the values to use for all conditional inputs. This must be specified if the GAN has any conditional inputs.
- **generator_index** (*int*) – the index of the generator (between 0 and n_generators-1) to use for generating the samples.

Returns

- *An array (if the generator has only one output) or list of arrays (if it has*
- *multiple outputs) containing the generated samples.*

WGAN

class WGAN(*gradient_penalty=10.0*, ***kwargs*)

Implements Wasserstein Generative Adversarial Networks.

This class implements Wasserstein Generative Adversarial Networks (WGANs) as described in Arjovsky et al., "Wasserstein GAN" (https://arxiv.org/abs/1701.07875). A WGAN is conceptually rather different from a conventional GAN, but in practical terms very similar. It reinterprets the discriminator (often called the "critic" in this context) as learning an approximation to the Earth Mover distance between the training and generated distributions. The generator is then trained to minimize that distance. In practice, this just means using slightly different loss functions for training the generator and discriminator.

WGANs have theoretical advantages over conventional GANs, and they often work better in practice. In addition, the discriminator's loss function can be directly interpreted as a measure of the quality of the model. That is an advantage over conventional GANs, where the loss does not directly convey information about the quality of the model.

The theory WGANs are based on requires the discriminator's gradient to be bounded. The original paper achieved this by clipping its weights. This class instead does it by adding a penalty term to the discriminator's loss, as described in https://arxiv.org/abs/1704.00028. This is sometimes found to produce better results.

There are a few other practical differences between GANs and WGANs. In a conventional GAN, the discriminator's output must be between 0 and 1 so it can be interpreted as a probability. In a WGAN, it should produce an unbounded output that can be interpreted as a distance.

When training a WGAN, you also should usually use a smaller value for generator_steps. Conventional GANs rely on keeping the generator and discriminator "in balance" with each other. If the discriminator ever gets too good, it becomes impossible for the generator to fool it and training stalls. WGANs do not have this problem, and in fact the better the discriminator is, the easier it is for the generator to improve. It therefore usually works best to perform several training steps on the discriminator for each training step on the generator.

`__init__`(*gradient_penalty=10.0*, ***kwargs*)

Construct a WGAN.

In addition to the following, this class accepts all the keyword arguments from GAN and KerasModel.

Parameters
`gradient_penalty` (*float*) – the magnitude of the gradient penalty loss

`create_generator_loss`(*discrim_output*)

Create the loss function for the generator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters
`discrim_output` (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Return type
A Tensor equal to the loss function to use for optimizing the generator.

`create_discriminator_loss`(*discrim_output_train*, *discrim_output_gen*)

Create the loss function for the discriminator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters

- **`discrim_output_train`** (*Tensor*) – the output from the discriminator on a batch of training data. This is its estimate of the probability that each sample is training data.
- **`discrim_output_gen`** (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Return type
A Tensor equal to the loss function to use for optimizing the discriminator.

3.17.16 TextCNNModel

class `TextCNNModel`(*n_tasks*, *char_dict*, *seq_length*, *n_embedding=75*, *kernel_sizes=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20]*, *num_filters=[100, 200, 200, 200, 200, 100, 100, 100, 100, 100, 160, 160]*, *dropout=0.25*, *mode='classification'*, ***kwargs*)

A Convolutional neural network on smiles strings

Reimplementation of the discriminator module in ORGAN **[1]_** . Originated from **[2]_**.

This model applies multiple 1D convolutional filters to the padded strings, then max-over-time pooling is applied on all filters, extracting one feature per filter. All features are concatenated and transformed through several hidden layers to form predictions.

This model is initially developed for sentence-level classification tasks, with words represented as vectors. In this implementation, SMILES strings are dissected into characters and transformed to one-hot vectors in a similar way. The model can be used for general molecular-level classification or regression tasks. It is also used in the ORGAN model as discriminator.

Training of the model only requires SMILES strings input, all featurized datasets that include SMILES in the *ids* attribute are accepted. PDBbind, QM7 and QM7b are not supported. To use the model, *build_char_dict* should be called first before defining the model to build character dict of input dataset, example can be found in examples/delaney/delaney_textcnn.py

References

`__init__`(*n_tasks*, *char_dict*, *seq_length*, *n_embedding=75*, *kernel_sizes=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20]*, *num_filters=[100, 200, 200, 200, 200, 100, 100, 100, 100, 100, 160, 160]*, *dropout=0.25*, *mode='classification'*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **char_dict** (*dict*) – Mapping from characters in smiles to integers
- **seq_length** (*int*) – Length of sequences(after padding)
- **n_embedding** (*int, optional*) – Length of embedding vector
- **filter_sizes** (*list of int, optional*) – Properties of filters used in the conv net
- **num_filters** (*list of int, optional*) – Properties of filters used in the conv net
- **dropout** (*float, optional*) – Dropout rate
- **mode** (*str*) – Either “classification” or “regression” for type of model.

static build_char_dict(*dataset*, *default_dict={'#': 1, '(': 2, ')': 3, '+': 4, '-': 5, '/': 6, '1': 7, '2': 8, '3': 9, '4': 10, '5': 11, '6': 12, '7': 13, '8': 14, '=': 15, 'Br': 30, 'C': 16, 'Cl': 29, 'F': 17, 'H': 18, 'I': 19, 'N': 20, 'O': 21, 'P': 22, 'S': 23, '[': 24, '\\': 25, ']': 26, '_': 27, 'c': 28, 'n': 31, 'o': 32, 's': 33}*)

Collect all unique characters(in smiles) from the dataset. This method should be called before defining the model to build appropriate char_dict

smiles_to_seq_batch(*ids_b*)

Converts SMILES strings to np.array sequence.

A tf.py_func wrapper is written around this when creating the input_fn for make_estimator

default_generator(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Transfer smiles strings to fixed length integer vectors

smiles_to_seq(*smiles*)

Tokenize characters in smiles to integers

3.17.17 AtomicConvModel

class AtomicConvModel(*n_tasks: int, frag1_num_atoms: int = 70, frag2_num_atoms: int = 634, complex_num_atoms: int = 701, max_num_neighbors: int = 12, batch_size: int = 24, atom_types: ~typing.Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0], radial: ~typing.Sequence[~typing.Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]], layer_sizes=[100], weight_init_stddevs: float | ~typing.Sequence[float] = 0.02, bias_init_consts: float | ~typing.Sequence[float] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.5, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = <function relu>, residual: bool = False, learning_rate=0.001, **kwargs*)

Implements an Atomic Convolution Model.

Implements the atomic convolutional networks as introduced in

Gomes, Joseph, et al. "Atomic convolutional networks for predicting protein-ligand binding affinity." arXiv preprint arXiv:1703.10603 (2017).

The atomic convolutional networks function as a variant of graph convolutions. The difference is that the "graph" here is the nearest neighbors graph in 3D space. The AtomicConvModel leverages these connections in 3D space to train models that learn to predict energetic state starting from the spatial geometry of the model.

`__init__`(*n_tasks: int, frag1_num_atoms: int = 70, frag2_num_atoms: int = 634, complex_num_atoms: int = 701, max_num_neighbors: int = 12, batch_size: int = 24, atom_types: ~typing.Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0], radial: ~typing.Sequence[~typing.Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]], layer_sizes=[100], weight_init_stddevs: float | ~typing.Sequence[float] = 0.02, bias_init_consts: float | ~typing.Sequence[float] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.5, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = <function relu>, residual: bool = False, learning_rate=0.001, **kwargs*) → None

Parameters

- **n_tasks** (*int*) – number of tasks
- **frag1_num_atoms** (*int*) – Number of atoms in first fragment
- **frag2_num_atoms** (*int*) – Number of atoms in sec
- **max_num_neighbors** (*int*) – Maximum number of neighbors possible for an atom. Recall neighbors are spatial neighbors.
- **atom_types** (*list*) – List of atoms recognized by model. Atoms are indicated by their nuclear numbers.
- **radial** (*list*) – Radial parameters used in the atomic convolution transformation.
- **layer_sizes** (*list*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or float*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float*) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str*) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float*) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (*list or object*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

- **residual** (*bool*) – if True, the model will be composed of pre-activation residual blocks instead of a simple stack of dense layers.
- **learning_rate** (*float*) – Learning rate for the model.

default_generator(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

save()

Saves model to disk using joblib.

reload()

Loads model from joblib file on disk.

3.17.18 Smiles2Vec

class Smiles2Vec(*char_to_idx*, *n_tasks=10*, *max_seq_len=270*, *embedding_dim=50*, *n_classes=2*, *use_bidir=True*, *use_conv=True*, *filters=192*, *kernel_size=3*, *strides=1*, *rnn_sizes=[224, 384]*, *rnn_types=['GRU', 'GRU']*, *mode='regression'*, ***kwargs*)

Implements the Smiles2Vec model, that learns neural representations of SMILES strings which can be used for downstream tasks.

The model is based on the description in Goh et al., "SMILES2vec: An Interpretable General-Purpose Deep Neural Network for Predicting Chemical Properties" (https://arxiv.org/pdf/1712.02034.pdf). The goal here is to take SMILES strings as inputs, turn them into vector representations which can then be used in predicting molecular properties.

The model consists of an Embedding layer that retrieves embeddings for each character in the SMILES string. These embeddings are learnt jointly with the rest of the model. The output from the embedding layer is a tensor of shape (batch_size, seq_len, embedding_dim). This tensor can optionally be fed through a 1D convolutional layer, before being passed to a series of RNN cells (optionally bidirectional). The final output from the RNN cells aims to have learnt the temporal dependencies in the SMILES string, and in turn information about the structure of the molecule, which is then used for molecular property prediction.

In the paper, the authors also train an explanation mask to endow the model with interpretability and gain insights into its decision making. This segment is currently not a part of this implementation as this was developed for the purpose of investigating a transfer learning protocol, ChemNet (which can be found at https://arxiv.org/abs/1712.02734).

`__init__`(*char_to_idx*, *n_tasks=10*, *max_seq_len=270*, *embedding_dim=50*, *n_classes=2*, *use_bidir=True*, *use_conv=True*, *filters=192*, *kernel_size=3*, *strides=1*, *rnn_sizes=[224, 384]*, *rnn_types=['GRU', 'GRU']*, *mode='regression'*, ***kwargs*)

Parameters

- **char_to_idx** (*dict,*) – char_to_idx contains character to index mapping for SMILES characters
- **embedding_dim** (*int, default 50*) – Size of character embeddings used.
- **use_bidir** (*bool, default True*) – Whether to use BiDirectional RNN Cells
- **use_conv** (*bool, default True*) – Whether to use a conv-layer
- **kernel_size** (*int, default 3*) – Kernel size for convolutions
- **filters** (*int, default 192*) – Number of filters
- **strides** (*int, default 1*) – Strides used in convolution
- **rnn_sizes** (*list[int], default [224, 384]*) – Number of hidden units in the RNN cells
- **mode** (*str, default regression*) – Whether to use model for regression or classification

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.17.19 ChemCeption

class ChemCeption(*img_spec: str = 'std'*, *img_size: int = 80*, *base_filters: int = 16*, *inception_blocks: Dict = {'A': 3, 'B': 3, 'C': 3}*, *n_tasks: int = 10*, *n_classes: int = 2*, *augment: bool = False*, *mode: str = 'regression'*, ***kwargs*)

Implements the ChemCeption model that leverages the representational capacities of convolutional neural networks (CNNs) to predict molecular properties.

The model is based on the description in Goh et al., "Chemception: A Deep Neural Network with Minimal Chemistry Knowledge Matches the Performance of Expert-developed QSAR/QSPR Models" (https://arxiv.org/pdf/1706.06689.pdf). The authors use an image based representation of the molecule, where pixels encode different atomic and bond properties. More details on the image repres- entations can be found at https://arxiv.org/abs/1710.02238

The model consists of a Stem Layer that reduces the image resolution for the layers to follow. The output of the Stem Layer is followed by a series of Inception-Resnet blocks & a Reduction layer. Layers in the Inception-Resnet blocks process image tensors at multiple resolutions and use a ResNet style skip-connection, combining features from different resolutions. The Reduction layers reduce the spatial extent of the image by max-pooling and 2-strided convolutions. More details on these layers can be found in the ChemCeption paper referenced above. The output of the final Reduction layer is subject to a Global Average Pooling, and a fully-connected layer maps the features to downstream outputs.

In the ChemCeption paper, the authors perform real-time image augmentation by rotating images between 0 to 180 degrees. This can be done during model training by setting the augment argument to True.

`__init__`(*img_spec: str = 'std'*, *img_size: int = 80*, *base_filters: int = 16*, *inception_blocks: Dict = {'A': 3, 'B': 3, 'C': 3}*, *n_tasks: int = 10*, *n_classes: int = 2*, *augment: bool = False*, *mode: str = 'regression'*, ***kwargs*)

Parameters

- **img_spec** (*str, default std*) – Image specification used
- **img_size** (*int, default 80*) – Image size used
- **base_filters** (*int, default 16*) – Base filters used for the different inception and reduction layers
- **inception_blocks** (*dict,*) – Dictionary containing number of blocks for every inception layer
- **n_tasks** (*int, default 10*) – Number of classification or regression tasks
- **n_classes** (*int, default 2*) – Number of classes (used only for classification)
- **augment** (*bool, default False*) – Whether to augment images
- **mode** (*str, default regression*) – Whether the model is used for regression or classification

`build_inception_module`(*inputs*, *type='A'*)

Inception module is a series of inception layers of similar type. This function builds that.

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (`Dataset`) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*

- *([inputs], [outputs], [weights])*

3.17.20 NormalizingFlowModel

The purpose of a normalizing flow is to map a simple distribution (that is easy to sample from and evaluate probability densities for) to a more complex distribution that is learned from data. Normalizing flows combine the advantages of autoregressive models (which provide likelihood estimation but do not learn features) and variational autoencoders (which learn feature representations but do not provide marginal likelihoods). They are effective for any application requiring a probabilistic model with these capabilities, e.g. generative modeling, unsupervised learning, or probabilistic inference.

`class NormalizingFlowModel`(*model: NormalizingFlow*, ***kwargs*)

A base distribution and normalizing flow for applying transformations.

Normalizing flows are effective for any application requiring a probabilistic model that can both sample from a distribution and compute marginal likelihoods, e.g. generative modeling, unsupervised learning, or probabilistic inference. For a thorough review of normalizing flows, see **[1]_**.

A distribution implements two main operations:

1. Sampling from the transformed distribution
2. Calculating log probabilities

A normalizing flow implements three main operations:

1. Forward transformation
2. Inverse transformation
3. Calculating the Jacobian

Deep Normalizing Flow models require normalizing flow layers where input and output dimensions are the same, the transformation is invertible, and the determinant of the Jacobian is efficient to compute and differentiable. The determinant of the Jacobian of the transformation gives a factor that preserves the probability volume to 1 when transforming between probability densities of different random variables.

References

`__init__`(*model: NormalizingFlow*, ***kwargs*) → None

Creates a new NormalizingFlowModel.

In addition to the following arguments, this class also accepts all the keyword arguments from KerasModel.

Parameters
`model` (`NormalizingFlow`) – An instance of NormalizingFlow.

Examples

>> import tensorflow_probability as tfp >> tfd = tfp.distributions >> tfb = tfp.bijectors >> flow_layers = [.. tfb.RealNVP(.. num_masked=2, .. shift_and_log_scale_fn=tfb.real_nvp_default_template(.. hidden_layers=[8, 8])) ..] >> base_distribution = tfd.MultivariateNormalDiag(loc=[0., 0., 0.]) >> nf = NormalizingFlow(base_distribution, flow_layers) >> nfm = NormalizingFlowModel(nf) >> dataset = NumpyDataset(.. X=np.random.rand(5, 3).astype(np.float32), .. y=np.random.rand(5,), .. ids=np.arange(5)) >> nfm.fit(dataset)

create_nll(*input: Tensor | Sequence[Tensor]*) → Tensor

Create the negative log likelihood loss function.

The default implementation is appropriate for most cases. Subclasses can override this if there is a need to customize it.

Parameters

input (*OneOrMany[tf.Tensor]*) – A batch of data.

Return type

A Tensor equal to the loss function to use for optimization.

save()

Saves model to disk using joblib.

reload()

Loads model from joblib file on disk.

3.18 PyTorch Models

DeepChem supports the use of PyTorch to build deep learning models.

3.18.1 TorchModel

You can wrap an arbitrary `torch.nn.Module` in a `TorchModel` object.

class TorchModel(*model: Module*, *loss: Loss | Callable[[List, List, List], Any]*, *output_types: List[str] | None = None*, *batch_size: int = 100*, *model_dir: str | None = None*, *learning_rate: float | LearningRateSchedule = 0.001*, *optimizer: Optimizer | None = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *device: device | None = None*, *regularization_loss: Callable | None = None*, *wandb_logger: WandbLogger | None = None*, ***kwargs*)

This is a DeepChem model implemented by a PyTorch model.

Here is a simple example of code that uses TorchModel to train a PyTorch model on a DeepChem dataset.

```
>>> import torch
>>> import deepchem as dc
>>> import numpy as np
>>> X, y = np.random.random((10, 100)), np.random.random((10, 1))
>>> dataset = dc.data.NumpyDataset(X=X, y=y)
>>> pytorch_model = torch.nn.Sequential(
...   torch.nn.Linear(100, 1000),
...   torch.nn.Tanh(),
...   torch.nn.Linear(1000, 1))
>>> model = dc.models.TorchModel(pytorch_model, loss=dc.models.losses.L2Loss())
>>> loss = model.fit(dataset, nb_epoch=5)
```

The loss function for a model can be defined in two different ways. For models that have only a single output and use a standard loss function, you can simply provide a dc.models.losses.Loss object. This defines the loss for each sample or sample/task pair. The result is automatically multiplied by the weights and averaged over the batch.

For more complicated cases, you can instead provide a function that directly computes the total loss. It must be of the form f(outputs, labels, weights), taking the list of outputs from the model, the expected values, and any weight matrices. It should return a scalar equal to the value of the loss function for the batch. No additional processing is done to the result; it is up to you to do any weighting, averaging, adding of penalty terms, etc.

You can optionally provide an output_types argument, which describes how to interpret the model's outputs. This should be a list of strings, one for each output. You can use an arbitrary output_type for a output, but some output_types are special and will undergo extra processing:

- **'prediction': This is a normal output, and will be returned by predict().**
  If output types are not specified, all outputs are assumed to be of this type.
- **'loss': This output will be used in place of the normal**
  outputs for computing the loss function. For example, models that output probability distributions usually do it by computing unbounded numbers (the logits), then passing them through a softmax function to turn them into probabilities. When computing the cross entropy, it is more numerically stable to use the logits directly rather than the probabilities. You can do this by having the model produce both probabilities and logits as outputs, then specifying output_types=['prediction', 'loss']. When predict() is called, only the first output (the probabilities) will be returned. But during training, it is the second output (the logits) that will be passed to the loss function.
- **'variance': This output is used for estimating the**
  uncertainty in another output. To create a model that can estimate uncertainty, there must be the same number of 'prediction' and 'variance' outputs. Each variance output must have the same shape as the corresponding prediction output, and each element is an estimate of the variance in the corresponding prediction. Also be aware that if a model supports uncertainty, it MUST use dropout on every layer, and dropout most be enabled during uncertainty prediction. Otherwise, the uncertainties it computes will be inaccurate.
- **other: Arbitrary output_types can be used to extract outputs**
  produced by the model, but will have no additional processing performed.

`__init__`(*model: Module*, *loss: Loss | Callable[[List, List, List], Any]*, *output_types: List[str] | None = None*, *batch_size: int = 100*, *model_dir: str | None = None*, *learning_rate: float | LearningRateSchedule = 0.001*, *optimizer: Optimizer | None = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *device: device | None = None*, *regularization_loss: Callable | None = None*, *wandb_logger: WandbLogger | None = None*, ***kwargs*) → None

Create a new TorchModel.

Parameters

- `model` (*torch.nn.Module*) – the PyTorch model implementing the calculation
- `loss` (*dc.models.losses.Loss or function*) – a Loss or function defining how to compute the training loss for each batch, as described above
- `output_types` (*list of strings, optional (default None)*) – the type of each output from the model, as described above
- `batch_size` (*int, optional (default 100)*) – default batch size for training and evaluating
- `model_dir` (*str, optional (default None)*) – the directory on disk where the model will be stored. If this is None, a temporary directory is created.
- `learning_rate` (*float or LearningRateSchedule, optional (default 0.001)*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.
- `optimizer` (*Optimizer, optional (default None)*) – the optimizer to use for fitting. If this is specified, learning_rate is ignored.
- `tensorboard` (*bool, optional (default False)*) – whether to log progress to TensorBoard during training

- **wandb** (*bool, optional (default False)*) – whether to log progress to Weights & Biases during training
- **log_frequency** (*int, optional (default 100)*) – The frequency at which to log data. Data is logged using *logging* by default. If *tensorboard* is set, data is also logged to TensorBoard. If *wandb* is set, data is also logged to Weights & Biases. Logging happens at global steps. Roughly, a global step corresponds to one batch of training. If you'd like a printout every 10 batch steps, you'd set *log_frequency=10* for example.
- **device** (*torch.device, optional (default None)*) – the device on which to run computations. If None, a device is chosen automatically.
- **regularization_loss** (*Callable, optional*) – a function that takes no arguments, and returns an extra contribution to add to the loss function
- **wandb_logger** (*WandbLogger*) – the Weights & Biases logger object used to log data and metrics

fit(*dataset: Dataset*, *nb_epoch: int = 10*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *deterministic: bool = False*, *restore: bool = False*, *variables: List[Parameter] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on a dataset.

Parameters

- **dataset** (*Dataset*) – the Dataset to train on
- **nb_epoch** (*int*) – the number of epochs to train for
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **deterministic** (*bool*) – if True, the samples are processed in order. If False, a different random order is used for each epoch.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step, **kwargs) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type

The average loss over the most recent checkpoint interval

fit_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *restore: bool = False*, *variables: List[Parameter] | ParameterList | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on data from a generator.

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of torch.nn.Parameter or torch.nn.ParameterList*) – the variables to train. If None (the default), all trainable variables in the model are used. ParameterList can be used like a regular Python list, but Tensors that are *Parameter* are properly registered, and will be visible by all *Module* methods.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type

The average loss over the most recent checkpoint interval

fit_on_batch(*X: Sequence*, *y: Sequence*, *w: Sequence*, *variables: List[Parameter] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *checkpoint: bool = True*, *max_checkpoints_to_keep: int = 5*) → float

Perform a single step of training.

Parameters

- **X** (*ndarray*) – the inputs for the batch
- **y** (*ndarray*) – the labels for the batch
- **w** (*ndarray*) – the weights for the batch
- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **checkpoint** (*bool*) – if true, save a checkpoint after performing the training step
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.

Return type
the loss on the batch

predict_on_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[*Transformer*] = []*, *output_types: str | Sequence[str] | None = None*) → ndarray | Sequence[ndarray]

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.
- **Returns** – a NumPy array of the model produces a single output, or a list of arrays if it produces multiple outputs

predict_on_batch(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *transformers: List[*Transformer*] = []*) → ndarray | Sequence[ndarray]

Generates predictions for input samples, processing samples in a batch.

Parameters

- **X** (*ndarray*) – the input data, as a Numpy array.
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

predict_uncertainty_on_batch(*X: Sequence*, *masks: int = 50*) → Tuple[ndarray, ndarray] | Sequence[Tuple[ndarray, ndarray]]

Predict the model's outputs, along with the uncertainty in each one.

The uncertainty is computed as described in https://arxiv.org/abs/1703.04977. It involves repeating the prediction many times with different dropout masks. The prediction is computed as the average over all the predictions. The uncertainty includes both the variation among the predicted values (epistemic uncertainty) and the model's own estimates for how well it fits the data (aleatoric uncertainty). Not all models support uncertainty prediction.

Parameters

- **X** (*ndarray*) – the input data, as a Numpy array.
- **masks** (*int*) – the number of dropout masks to average over

Returns

- *for each output, a tuple (y_pred, y_std) where y_pred is the predicted*
- *value of the output, and each element of y_std estimates the standard*
- *deviation of the corresponding element of y_pred*

predict(*dataset: Dataset*, *transformers: List[Transformer] = []*, *output_types: List[str] | None = None*) → ndarray | Sequence[ndarray]

Uses self to make predictions on provided Dataset object.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to make prediction on
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

predict_embedding(*dataset: Dataset*) → ndarray | Sequence[ndarray]

Predicts embeddings created by underlying model if any exist. An embedding must be specified to have *output_type* of *'embedding'* in the model definition.

Parameters
dataset (*dc.data.Dataset*) – Dataset to make prediction on

Returns

- *a NumPy array of the embeddings model produces, or a list*
- *of arrays if it produces multiple embeddings*

predict_uncertainty(*dataset: Dataset*, *masks: int = 50*) → Tuple[ndarray, ndarray] | Sequence[Tuple[ndarray, ndarray]]

Predict the model's outputs, along with the uncertainty in each one.

The uncertainty is computed as described in https://arxiv.org/abs/1703.04977. It involves repeating the prediction many times with different dropout masks. The prediction is computed as the average over all the predictions. The uncertainty includes both the variation among the predicted values (epistemic uncertainty) and the model's own estimates for how well it fits the data (aleatoric uncertainty). Not all models support uncertainty prediction.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to make prediction on
- **masks** (*int*) – the number of dropout masks to average over

Returns

- *for each output, a tuple (y_pred, y_std) where y_pred is the predicted*
- *value of the output, and each element of y_std estimates the standard*
- *deviation of the corresponding element of y_pred*

evaluate_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *metrics: List[Metric]*, *transformers: List[Transformer] = []*, *per_task_metrics: bool = False*)

Evaluate the performance of this model on the data produced by a generator.

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **metric** (*list of* deepchem.metrics.Metric) – Evaluation metric
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **per_task_metrics** (*bool*) – If True, return per-task scores.

Returns
Maps tasks to scores under metric.

Return type
dict

compute_saliency(*X: ndarray*) → ndarray | Sequence[ndarray]

Compute the saliency map for an input sample.

This computes the Jacobian matrix with the derivative of each output element with respect to each input element. More precisely,

- **If this model has a single output, it returns a matrix of shape**
  (output_shape, input_shape) with the derivatives.
- **If this model has multiple outputs, it returns a list of matrices, one**
  for each output.

This method cannot be used on models that take multiple inputs.

Parameters
X (*ndarray*) – the input data for a single sample

Return type
the Jacobian matrix, or a list of matrices

default_generator(*dataset:* Dataset, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

save_checkpoint(*max_checkpoints_to_keep: int = 5*, *model_dir: str | None = None*) → None

Save a checkpoint to disk.

Usually you do not need to call this method, since fit() saves checkpoints automatically. If you have disabled automatic checkpointing during fitting, this can be called to manually write checkpoints.

Parameters

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded. If set to zero, the function will simply return as no checkpoint is saved.
- **model_dir** (*str, default None*) – Model directory to save checkpoint to. If None, revert to self.model_dir

get_checkpoints(*model_dir: str | None = None*)

Get a list of all available checkpoint files.

Parameters

model_dir (*str, default None*) – Directory to get list of checkpoints from. Reverts to self.model_dir if None

restore(*checkpoint: str | None = None*, *model_dir: str | None = None*, *strict: bool | None = True*) → None

Reload the values of all variables from a checkpoint file.

Parameters

- **checkpoint** (*str*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints.
- **model_dir** (*str, default None*) – Directory to restore checkpoint from. If None, use self.model_dir. If checkpoint is not None, this is ignored.
- **strict** (*bool, default True*) – Whether or not to strictly enforce that the keys in checkpoint match the keys returned by this model's get_variable_scope() method.

get_global_step() → int

Get the number of steps of fitting that have been performed.

load_from_pretrained(*source_model: TorchModel*, *assignment_map: Dict[Any, Any] | None = None*, *value_map: Dict[Any, Any] | None = None*, *checkpoint: str | None = None*, *model_dir: str | None = None*, *include_top: bool = True*, *inputs: Sequence[Any] | None = None*, ***kwargs*) → None

Copies parameter values from a pretrained model. *source_model* can either be a pretrained model or a model with the same architecture. *value_map* is a parameter-value dictionary. If no *value_map* is provided, the parameter values are restored to the *source_model* from a checkpoint and a default *value_map* is created. *assignment_map* is a dictionary mapping parameters from the *source_model* to the current model. If no *assignment_map* is provided, one is made from scratch and assumes the model is composed of several different layers, with the final one being a dense layer. include_top is used to control whether or not the final dense layer is used. The default assignment map is useful in cases where the type of task is different (classification vs regression) and/or number of tasks in the setting.

Parameters

- **source_model** (*dc.TorchModel, required*) – source_model can either be the pre-trained model or a dc.TorchModel with the same architecture as the pretrained model. It is used to restore from a checkpoint, if value_map is None and to create a default assignment map if assignment_map is None

- **assignment_map** (*Dict, default None*) – Dictionary mapping the source_model parameters and current model parameters
- **value_map** (*Dict, default None*) – Dictionary containing source_model trainable parameters mapped to numpy arrays. If value_map is None, the values are restored and a default parameter map is created using the restored values
- **checkpoint** (*str, default None*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints
- **model_dir** (*str, default None*) – Restore source model from custom model directory if needed
- **include_top** (*bool, default True*) – if True, copies the weights and bias associated with the final dense layer. Used only when assignment map is None
- **inputs** (*List, input tensors for model*) – if not None, then the weights are built for both the source and self.

3.18.2 ModularTorchModel

You can modify networks for different tasks by using a `ModularTorchModel`.

class ModularTorchModel(*model: Module*, *components: dict*, ***kwargs*)

ModularTorchModel is a subclass of TorchModel that allows for components to be pretrained and then combined into a final model. It is designed to be subclassed for specific models and is not intended to be used directly. There are 3 main differences between ModularTorchModel and TorchModel:

- The build_components() method is used to define the components of the model.
- The components are combined into a final model with the build_model() method.
- The loss function is defined with the loss_func method. This may access the components to compute the loss using intermediate values from the network, rather than just the full forward pass output.

Here is an example of how to use ModularTorchModel to pretrain a linear layer, load it into another network and then finetune that network:

```
>>> import numpy as np
>>> import deepchem as dc
>>> import torch
>>> n_samples = 6
>>> n_feat = 3
>>> n_hidden = 2
>>> n_tasks = 6
>>> pt_tasks = 3
>>> X = np.random.rand(n_samples, n_feat)
>>> y_pretrain = np.zeros((n_samples, pt_tasks)).astype(np.float32)
>>> dataset_pt = dc.data.NumpyDataset(X, y_pretrain)
>>> y_finetune = np.zeros((n_samples, n_tasks)).astype(np.float32)
>>> dataset_ft = dc.data.NumpyDataset(X, y_finetune)
>>> components = {'linear': torch.nn.Linear(n_feat, n_hidden),
... 'activation': torch.nn.ReLU(), 'head': torch.nn.Linear(n_hidden, n_tasks)}
>>> model = torch.nn.Sequential(components['linear'], components['activation'],
... components['head'])
>>> modular_model = dc.models.torch_models.modular.ModularTorchModel(model,␣
```

(continues on next page)

(continued from previous page)

```
↪components)
>>> def example_loss_func(inputs, labels, weights):
...    return (torch.nn.functional.mse_loss(model(inputs), labels[0]) * weights[0]).
↪mean()
>>> modular_model.loss_func = example_loss_func
>>> def example_model_build():
...     return torch.nn.Sequential(components['linear'], components['activation'],
... components['head'])
>>> modular_model.build_model = example_model_build
>>> pretrain_components = {'linear': torch.nn.Linear(n_feat, n_hidden),
... 'activation': torch.nn.ReLU(), 'head': torch.nn.Linear(n_hidden, pt_tasks)}
>>> pretrain_model = torch.nn.Sequential(pretrain_components['linear'],
... pretrain_components['activation'], pretrain_components['head'])
>>> pretrain_modular_model = dc.models.torch_models.modular.
↪ModularTorchModel(pretrain_model,
... pretrain_components)
>>> def example_pt_loss_func(inputs, labels, weights):
...     return (torch.nn.functional.mse_loss(pretrain_model(inputs), labels[0]) *␣
↪weights[0]).mean()
>>> pretrain_modular_model.loss_func = example_pt_loss_func
>>> pt_loss = pretrain_modular_model.fit(dataset_pt, nb_epoch=1)
>>> modular_model.load_from_pretrained(pretrain_modular_model, components=['linear
↪'])
>>> ft_loss = modular_model.fit(dataset_ft, nb_epoch=1)
```

`__init__`(*model: Module*, *components: dict*, ***kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

`build_model`() → Module

Builds the final model from the components.

`build_components`() → dict

Creates the components dictionary, with the keys being the names of the components and the values being torch.nn.module objects.

`loss_func`(*inputs: Tensor | Sequence[Tensor]*, *labels: Sequence*, *weights: Sequence*) → Tensor

Defines the loss function for the model which can access the components using self.components. The loss function should take the inputs, labels, and weights as arguments and return the loss.

`freeze_components`(*components: List[str]*)

Freezes or unfreezes the parameters of the specified components.

Components string refers to keys in self.components.

Parameters

components (*List[str]*) – The components to freeze.

`unfreeze_components`(*components: List[str]*)

Unfreezes the parameters of the specified components.

Components string refers to keys in self.components.

Parameters
components (*List[str]*) – The components to unfreeze.

fit_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *restore: bool = False*, *variables: List[Parameter] | ParameterList | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on data from a generator. This method is similar to the TorchModel implementation, but it passes the inputs directly to the loss function, rather than passing them through the model first. This enables the loss to be calculated from intermediate steps of the model and not just the final output.

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step, **kwargs) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type
The average loss over the most recent checkpoint interval

load_from_pretrained(*source_model: ModularTorchModel | None = None*, *components: List[str] | None = None*, *checkpoint: str | None = None*, *model_dir: str | None = None*, *inputs: Sequence[Any] | None = None*, ***kwargs*) → None

Copies parameter values from a pretrained model. The pretrained model can be loaded as a source_model (ModularTorchModel object), checkpoint (pytorch .ckpt file) or a model_dir (directory with .ckpt files). Specific components can be chosen by passing a list of strings with the desired component names. If both a source_model and a checkpoint/model_dir are loaded, the source_model weights will be loaded.

Parameters

- **source_model** (*dc.ModularTorchModel, required*) – source_model can either be the pretrained model or a dc.TorchModel with the same architecture as the pretrained model. It is used to restore from a checkpoint, if value_map is None and to create a default assignment map if assignment_map is None

- **checkpoint** (*str, default None*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints
- **model_dir** (*str, default None*) – Restore source model from custom model directory if needed
- **inputs** (*List, input tensors for model*) – if not None, then the weights are built for both the source and self.

save_checkpoint(*max_checkpoints_to_keep=5*, *model_dir=None*)

Saves the current state of the model and its components as a checkpoint file in the specified model directory. It maintains a maximum number of checkpoint files, deleting the oldest one when the limit is reached.

Parameters

- **max_checkpoints_to_keep** (*int, default 5*) – Maximum number of checkpoint files to keep.
- **model_dir** (*str, default None*) – The directory to save the checkpoint file in. If None, the model_dir specified in the constructor is used.

restore(*components: List[str] | None = None*, *checkpoint: str | None = None*, *model_dir: str | None = None*) → None

Restores the state of a ModularTorchModel from a checkpoint file.

If no checkpoint file is provided, it will use the latest checkpoint found in the model directory. If a list of component names is provided, only the state of those components will be restored.

Parameters

- **components** (*Optional[List[str]]*) – A list of component names to restore. If None, all components will be restored.
- **checkpoint** (*Optional[str]*) – The path to the checkpoint file. If None, the latest checkpoint in the model directory will be used.
- **model_dir** (*Optional[str]*) – The path to the model directory. If None, the model directory used to initialize the model will be used.

3.18.3 CNN

class CNN(*n_tasks: int*, *n_features: int*, *dims: int*, *layer_filters: List[int] = [100]*, *kernel_size: int | Sequence[int] = 5*, *strides: int | Sequence[int] = 1*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *pool_type: str = 'max'*, *mode: str = 'classification'*, *n_classes: int = 2*, *uncertainty: bool = False*, *residual: bool = False*, *padding: int | str = 'valid'*, ***kwargs*)

A 1, 2, or 3 dimensional convolutional network for either regression or classification.

The network consists of the following sequence of layers:

- A configurable number of convolutional layers
- A global pooling layer (either max pool or average pool)
- A final fully connected layer to compute the output

It optionally can compose the model from pre-activation residual blocks, as described in https://arxiv.org/abs/1603.05027, rather than a simple stack of convolution layers. This often leads to easier training, especially when using a large number of layers. Note that residual blocks can only be used when successive layers have the same output shape. Wherever the output shape changes, a simple convolution layer will be used even if residual=True.

Examples

```
>>> import deepchem as dc
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> np.random.seed(123)
>>> X = np.random.rand(n_samples, 10, n_features)
>>> y = np.random.randint(2, size=(n_samples, n_tasks)).astype(np.float32)
>>> dataset: dc.data.Dataset = dc.data.NumpyDataset(X, y)
>>> regression_metric = dc.metrics.Metric(dc.metrics.mean_squared_error)
>>> model = CNN(n_tasks, n_features, dims=1, kernel_size=3, mode='regression')
>>> avg_loss = model.fit(dataset, nb_epoch=10)
```

`__init__`(*n_tasks: int*, *n_features: int*, *dims: int*, *layer_filters: List[int] = [100]*, *kernel_size: int | Sequence[int] = 5*, *strides: int | Sequence[int] = 1*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *pool_type: str = 'max'*, *mode: str = 'classification'*, *n_classes: int = 2*, *uncertainty: bool = False*, *residual: bool = False*, *padding: int | str = 'valid'*, ***kwargs*) → None

TorchModel wrapper for CNN

Parameters

- **n_tasks** (*int*) – number of tasks
- **n_features** (*int*) – number of features
- **dims** (*int*) – the number of dimensions to apply convolutions over (1, 2, or 3)
- **layer_filters** (*list*) – the number of output filters for each convolutional layer in the network. The length of this list determines the number of layers.
- **kernel_size** (*int, tuple, or list*) – a list giving the shape of the convolutional kernel for each layer. Each element may be either an int (use the same kernel width for every dimension) or a tuple (the kernel width along each dimension). Alternatively this may be a single int or tuple instead of a list, in which case the same kernel shape is used for every layer.
- **strides** (*int, tuple, or list*) – a list giving the stride between applications of the kernel for each layer. Each element may be either an int (use the same stride for every dimension) or a tuple (the stride along each dimension). Alternatively this may be a single int or tuple instead of a list, in which case the same stride is used for every layer.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_filters)+1, where the final element corresponds to the dense layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or float*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_filters)+1, where the final element corresponds

to the dense layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

- **weight_decay_penalty** (*float*) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str*) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float*) – the dropout probability to use for each layer. The length of this list should equal len(layer_filters). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer
- **activation_fns** (*str or list*) – the torch activation function to apply to each layer. The length of this list should equal len(layer_filters). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer, 'relu' by default
- **pool_type** (*str*) – the type of pooling layer to use, either 'max' or 'average'
- **mode** (*str*) – Either 'classification' or 'regression'
- **n_classes** (*int*) – the number of classes to predict (only used in classification mode)
- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted
- **residual** (*bool*) – if True, the model will be composed of pre-activation residual blocks instead of a simple stack of convolutional layers.
- **padding** (*str, int or tuple*) – the padding to use for convolutional layers, either 'valid' or 'same'

default_generator(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.4 MultitaskRegressor

class MultitaskRegressor(*n_tasks: int*, *n_features: int*, *layer_sizes: Sequence[int] = [1000]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *uncertainty: bool = False*, *residual: bool = False*, ***kwargs*)

A fully connected network for multitask regression.

This class provides lots of options for customizing aspects of the model: the number and widths of layers, the activation functions, regularization methods, etc.

It optionally can compose the model from pre-activation residual blocks, as described in https://arxiv.org/abs/1603.05027, rather than a simple stack of dense layers. This often leads to easier training, especially when using a large number of layers. Note that residual blocks can only be used when successive layers have the same width. Wherever the layer width changes, a simple dense layer will be used even if residual=True.

__init__(*n_tasks: int*, *n_features: int*, *layer_sizes: Sequence[int] = [1000]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *uncertainty: bool = False*, *residual: bool = False*, ***kwargs*) → None

Create a MultitaskRegressor.

In addition to the following arguments, this class also accepts all the keywork arguments from TensorGraph.

Parameters

- **n_tasks** (`int`) – number of tasks
- **n_features** (`int`) – number of features
- **layer_sizes** (`list`) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (`list or float`) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (`list or float`) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes)+1. The final element corresponds to the output layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (`float`) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (`str`) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (`list or float`) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (`list or object`) – the PyTorch activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer. Standard activation functions from torch.nn.functional can be specified by name.

- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted
- **residual** (*bool*) – if True, the model will be composed of pre-activation residual blocks instead of a simple stack of dense layers.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.5 MultitaskFitTransformRegressor

class MultitaskFitTransformRegressor(*n_tasks: int*, *n_features: int*, *fit_transformers: Sequence[Transformer] = []*, *batch_size: int = 50*, ***kwargs*)

Implements a MultitaskRegressor that performs on-the-fly transformation during fit/predict.

Examples

```
>>> n_samples = 10
>>> n_features = 3
>>> n_tasks = 1
>>> ids = np.arange(n_samples)
>>> X = np.random.rand(n_samples, n_features, n_features)
>>> y = np.zeros((n_samples, n_tasks))
>>> w = np.ones((n_samples, n_tasks))
>>> dataset = dc.data.NumpyDataset(X, y, w, ids)
>>> fit_transformers = [dc.trans.CoulombFitTransformer(dataset)]
>>> model = dc.models.MultitaskFitTransformRegressor(n_tasks, [n_features, n_
↪features],
...     dropouts=[0.], learning_rate=0.003, weight_init_stddevs=[np.sqrt(6)/np.
↪sqrt(1000)],
...     batch_size=n_samples, fit_transformers=fit_transformers)
>>> model.n_features
12
```

`__init__`(*n_tasks: int*, *n_features: int*, *fit_transformers: Sequence[*Transformer*] = []*, *batch_size: int = 50*, ***kwargs*)

Create a MultitaskFitTransformRegressor.

In addition to the following arguments, this class also accepts all the keywork arguments from MultitaskRegressor.

Parameters

- **n_tasks** (*int*) – number of tasks
- **n_features** (*list or int*) – number of features
- **fit_transformers** (*list*) – List of dc.trans.FitTransformer objects

`default_generator`(*dataset:* Dataset, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

`predict_on_generator`(*generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[*Transformer*] = []*, *output_types: str | Sequence[str] | None = None*) → ndarray | Sequence[ndarray]

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **transformers** (*list of dc.trans.Transformers*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.
- **Returns** – a NumPy array of the model produces a single output, or a list of arrays if it produces multiple outputs

3.18.6 MultitaskClassifier

class MultitaskClassifier(*n_tasks: int*, *n_features: int*, *layer_sizes: Sequence[int] = [1000]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *n_classes: int = 2*, *residual: bool = False*, ***kwargs*)

A fully connected network for multitask classification.

This class provides lots of options for customizing aspects of the model: the number and widths of layers, the activation functions, regularization methods, etc.

It optionally can compose the model from pre-activation residual blocks, as described in https://arxiv.org/abs/1603.05027, rather than a simple stack of dense layers. This often leads to easier training, especially when using a large number of layers. Note that residual blocks can only be used when successive layers have the same width. Wherever the layer width changes, a simple dense layer will be used even if residual=True.

__init__(*n_tasks: int*, *n_features: int*, *layer_sizes: Sequence[int] = [1000]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *n_classes: int = 2*, *residual: bool = False*, ***kwargs*) → None

Create a MultitaskClassifier.

In addition to the following arguments, this class also accepts all the keyword arguments from TensorGraph.

Parameters

- **n_tasks** (`int`) – number of tasks
- **n_features** (`int`) – number of features
- **layer_sizes** (`list`) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **weight_init_stddevs** (`list or float`) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (`list or float`) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (`float`) – the magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (`str`) – the type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (`list or float`) – the dropout probablity to use for each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **activation_fns** (`list or object`) – the PyTorch activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer. Standard activation functions from torch.nn.functional can be specified by name.
- **n_classes** (`int`) – the number of classes

- **residual** (*bool*) – if True, the model will be composed of pre-activation residual blocks instead of a simple stack of dense layers.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.7 CGCNNModel

class CGCNNModel(*in_node_dim: int = 92*, *hidden_node_dim: int = 64*, *in_edge_dim: int = 41*, *num_conv: int = 3*, *predictor_hidden_feats: int = 128*, *n_tasks: int = 1*, *mode: str = 'regression'*, *n_classes: int = 2*, ***kwargs*)

Crystal Graph Convolutional Neural Network (CGCNN).

Here is a simple example of code that uses the CGCNNModel with materials dataset.

Examples

```
>>> import deepchem as dc
>>> dataset_config = {"reload": False, "featurizer": dc.feat.CGCNNFeaturizer(),
↪"transformers": []}
>>> tasks, datasets, transformers = dc.molnet.load_perovskite(**dataset_config)
>>> train, valid, test = datasets
>>> model = dc.models.CGCNNModel(mode='regression', batch_size=32, learning_rate=0.
↪001)
>>> avg_loss = model.fit(train, nb_epoch=50)
```

This model takes arbitary crystal structures as an input, and predict material properties using the element information and connection of atoms in the crystal. If you want to get some material properties which has a high computational cost like band gap in the case of DFT, this model may be useful. This model is one of variants of Graph Convolutional Networks. The main differences between other GCN models are how to construct graphs and how to update node representations. This model defines the crystal graph from structures using distances between atoms. The crystal graph is an undirected multigraph which is defined by nodes representing atom properties and edges representing connections between atoms in a crystal. And, this model updates the node representations using both neighbor node and edge representations. Please confirm the detail algorithms from [1]_.

References

Notes

This class requires DGL and PyTorch to be installed.

`__init__`(*in_node_dim: int = 92*, *hidden_node_dim: int = 64*, *in_edge_dim: int = 41*, *num_conv: int = 3*, *predictor_hidden_feats: int = 128*, *n_tasks: int = 1*, *mode: str = 'regression'*, *n_classes: int = 2*, ***kwargs*)

This class accepts all the keyword arguments from TorchModel.

Parameters

- **in_node_dim** (*int, default 92*) – The length of the initial node feature vectors. The 92 is based on length of vectors in the atom_init.json.
- **hidden_node_dim** (*int, default 64*) – The length of the hidden node feature vectors.
- **in_edge_dim** (*int, default 41*) – The length of the initial edge feature vectors. The 41 is based on default setting of CGCNNFeaturizer.
- **num_conv** (*int, default 3*) – The number of convolutional layers.
- **predictor_hidden_feats** (*int, default 128*) – The size for hidden representations in the output MLP predictor.
- **n_tasks** (*int, default 1*) – The number of the output size.
- **mode** (*str, default 'regression'*) – The model type, 'classification' or 'regression'.
- **n_classes** (*int, default 2*) – The number of classes to predict (only used in classification mode).
- **kwargs** (*Dict*) – This class accepts all the keyword arguments from TorchModel.

3.18.8 GATModel

class GATModel(*n_tasks: int*, *graph_attention_layers: list | None = None*, *n_attention_heads: int = 8*, *agg_modes: list | None = None*, *activation=<function elu>*, *residual: bool = True*, *dropout: float = 0.0*, *alpha: float = 0.2*, *predictor_hidden_feats: int = 128*, *predictor_dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Model for Graph Property Prediction Based on Graph Attention Networks (GAT).

This model proceeds as follows:

- Update node representations in graphs with a variant of GAT
- **For each graph, compute its representation by 1) a weighted sum of the node**
  representations in the graph, where the weights are computed by applying a gating function to the node representations 2) a max pooling of the node representations 3) concatenating the output of 1) and 2)
- Perform the final prediction using an MLP

Examples

```
>>> import deepchem as dc
>>> from deepchem.models import GATModel
>>> # preparing dataset
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> labels = [0., 1.]
>>> featurizer = dc.feat.MolGraphConvFeaturizer()
>>> X = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(X=X, y=labels)
>>> # training model
>>> model = GATModel(mode='classification', n_tasks=1,
...                  batch_size=16, learning_rate=0.001)
>>> loss = model.fit(dataset, nb_epoch=5)
```

References

Notes

This class requires DGL (https://github.com/dmlc/dgl) and DGL-LifeSci (https://github.com/awslabs/dgl-lifesci) to be installed.

`__init__`(*n_tasks: int*, *graph_attention_layers: list | None = None*, *n_attention_heads: int = 8*, *agg_modes: list | None = None*, *activation=<function elu>*, *residual: bool = True*, *dropout: float = 0.0*, *alpha: float = 0.2*, *predictor_hidden_feats: int = 128*, *predictor_dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.
- **graph_attention_layers** (*list of int*) – Width of channels per attention head for GAT layers. graph_attention_layers[i] gives the width of channel for each attention head for the i-th GAT layer. If both `graph_attention_layers` and `agg_modes` are specified, they should have equal length. If not specified, the default value will be [8, 8].
- **n_attention_heads** (*int*) – Number of attention heads in each GAT layer.
- **agg_modes** (*list of str*) – The way to aggregate multi-head attention results for each GAT layer, which can be either 'flatten' for concatenating all-head results or 'mean' for averaging all-head results. `agg_modes[i]` gives the way to aggregate multi-head attention results for the i-th GAT layer. If both `graph_attention_layers` and `agg_modes` are specified, they should have equal length. If not specified, the model will flatten multi-head results for intermediate GAT layers and compute mean of multi-head results for the last GAT layer.
- **activation** (*activation function or None*) – The activation function to apply to the aggregated multi-head results for each GAT layer. If not specified, the default value will be ELU.
- **residual** (*bool*) – Whether to add a residual connection within each GAT layer. Default to True.
- **dropout** (*float*) – The dropout probability within each GAT layer. Default to 0.

- **alpha** (*float*) – A hyperparameter in LeakyReLU, which is the slope for negative values. Default to 0.2.
- **predictor_hidden_feats** (*int*) – The size for hidden representations in the output MLP predictor. Default to 128.
- **predictor_dropout** (*float*) – The dropout probability in the output MLP predictor. Default to 0.
- **mode** (*str*) – The model type, 'classification' or 'regression'. Default to 'regression'.
- **number_atom_features** (*int*) – The length of the initial atom feature vectors. Default to 30.
- **n_classes** (*int*) – The number of classes to predict per task (only used when `mode` is 'classification'). Default to 2.
- **self_loop** (*bool*) – Whether to add self loops for the nodes, i.e. edges from nodes to themselves. When input graphs have isolated nodes, self loops allow preserving the original feature of them in message passing. Default to True.
- **kwargs** – This can include any keyword argument of TorchModel.

3.18.9 GCNModel

class GCNModel(*n_tasks: int*, *graph_conv_layers: list | None = None*, *activation=None*, *residual: bool = True*, *batchnorm: bool = False*, *dropout: float = 0.0*, *predictor_hidden_feats: int = 128*, *predictor_dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features=30*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Model for Graph Property Prediction Based on Graph Convolution Networks (GCN).

This model proceeds as follows:

- Update node representations in graphs with a variant of GCN
- **For each graph, compute its representation by 1) a weighted sum of the node**
  representations in the graph, where the weights are computed by applying a gating function to the node representations 2) a max pooling of the node representations 3) concatenating the output of 1) and 2)
- Perform the final prediction using an MLP

Examples

```
>>> import deepchem as dc
>>> from deepchem.models import GCNModel
>>> # preparing dataset
>>> smiles = ["C1CCC1", "CCC"]
>>> labels = [0., 1.]
>>> featurizer = dc.feat.MolGraphConvFeaturizer()
>>> X = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(X=X, y=labels)
>>> # training model
>>> model = GCNModel(mode='classification', n_tasks=1,
...                  batch_size=16, learning_rate=0.001)
>>> loss = model.fit(dataset, nb_epoch=5)
```

References

Notes

This class requires DGL (https://github.com/dmlc/dgl) and DGL-LifeSci (https://github.com/awslabs/dgl-lifesci) to be installed.

This model is different from deepchem.models.GraphConvModel as follows:

- **For each graph convolution, the learnable weight in this model is shared across all nodes.**
  `GraphConvModel` employs separate learnable weights for nodes of different degrees. A learnable weight is shared across all nodes of a particular degree.
- **For `GraphConvModel`, there is an additional GraphPool operation after each**
  graph convolution. The operation updates the representation of a node by applying an element-wise maximum over the representations of its neighbors and itself.
- **For computing graph-level representations, this model computes a weighted sum and an**
  element-wise maximum of the representations of all nodes in a graph and concatenates them. The node weights are obtained by using a linear/dense layer followd by a sigmoid function. For `GraphConvModel`, the sum over node representations is unweighted.
- **There are various minor differences in using dropout, skip connection and batch**
  normalization.

`__init__`(*n_tasks: int*, *graph_conv_layers: list | None = None*, *activation=None*, *residual: bool = True*, *batchnorm: bool = False*, *dropout: float = 0.0*, *predictor_hidden_feats: int = 128*, *predictor_dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features=30*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Parameters

- **`n_tasks`** (*`int`*) – Number of tasks.
- **`graph_conv_layers`** (*`list of int`*) – Width of channels for GCN layers. graph_conv_layers[i] gives the width of channel for the i-th GCN layer. If not specified, the default value will be [64, 64].
- **`activation`** (*`callable`*) – The activation function to apply to the output of each GCN layer. By default, no activation function will be applied.
- **`residual`** (*`bool`*) – Whether to add a residual connection within each GCN layer. Default to True.
- **`batchnorm`** (*`bool`*) – Whether to apply batch normalization to the output of each GCN layer. Default to False.
- **`dropout`** (*`float`*) – The dropout probability for the output of each GCN layer. Default to 0.
- **`predictor_hidden_feats`** (*`int`*) – The size for hidden representations in the output MLP predictor. Default to 128.
- **`predictor_dropout`** (*`float`*) – The dropout probability in the output MLP predictor. Default to 0.
- **`mode`** (*`str`*) – The model type, 'classification' or 'regression'. Default to 'regression'.
- **`number_atom_features`** (*`int`*) – The length of the initial atom feature vectors. Default to 30.
- **`n_classes`** (*`int`*) – The number of classes to predict per task (only used when `mode` is 'classification'). Default to 2.

- **self_loop** (*bool*) – Whether to add self loops for the nodes, i.e. edges from nodes to themselves. When input graphs have isolated nodes, self loops allow preserving the original feature of them in message passing. Default to True.
- **kwargs** – This can include any keyword argument of TorchModel.

3.18.10 AttentiveFPModel

class AttentiveFPModel(*n_tasks: int*, *num_layers: int = 2*, *num_timesteps: int = 2*, *graph_feat_size: int = 200*, *dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *number_bond_features: int = 11*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Model for Graph Property Prediction.

This model proceeds as follows:

- **Combine node features and edge features for initializing node representations,**
  which involves a round of message passing
- Update node representations with multiple rounds of message passing
- **For each graph, compute its representation by combining the representations**
  of all nodes in it, which involves a gated recurrent unit (GRU).
- Perform the final prediction using a linear layer

Examples

```
>>> import deepchem as dc
>>> from deepchem.models import AttentiveFPModel
>>> # preparing dataset
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> labels = [0., 1.]
>>> featurizer = dc.feat.MolGraphConvFeaturizer(use_edges=True)
>>> X = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(X=X, y=labels)
>>> # training model
>>> model = AttentiveFPModel(mode='classification', n_tasks=1,
...    batch_size=16, learning_rate=0.001)
>>> loss = model.fit(dataset, nb_epoch=5)
```

References

Notes

This class requires DGL (https://github.com/dmlc/dgl) and DGL-LifeSci (https://github.com/awslabs/dgl-lifesci) to be installed.

__init__(*n_tasks: int*, *num_layers: int = 2*, *num_timesteps: int = 2*, *graph_feat_size: int = 200*, *dropout: float = 0.0*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *number_bond_features: int = 11*, *n_classes: int = 2*, *self_loop: bool = True*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.

- **num_layers** (*int*) – Number of graph neural network layers, i.e. number of rounds of message passing. Default to 2.
- **num_timesteps** (*int*) – Number of time steps for updating graph representations with a GRU. Default to 2.
- **graph_feat_size** (*int*) – Size for graph representations. Default to 200.
- **dropout** (*float*) – Dropout probability. Default to 0.
- **mode** (*str*) – The model type, 'classification' or 'regression'. Default to 'regression'.
- **number_atom_features** (*int*) – The length of the initial atom feature vectors. Default to 30.
- **number_bond_features** (*int*) – The length of the initial bond feature vectors. Default to 11.
- **n_classes** (*int*) – The number of classes to predict per task (only used when `mode` is 'classification'). Default to 2.
- **self_loop** (*bool*) – Whether to add self loops for the nodes, i.e. edges from nodes to themselves. When input graphs have isolated nodes, self loops allow preserving the original feature of them in message passing. Default to True.
- **kwargs** – This can include any keyword argument of TorchModel.

3.18.11 PagtnModel

class PagtnModel(*n_tasks: int*, *number_atom_features: int = 94*, *number_bond_features: int = 42*, *mode: str = 'regression'*, *n_classes: int = 2*, *output_node_features: int = 256*, *hidden_features: int = 32*, *num_layers: int = 5*, *num_heads: int = 1*, *dropout: float = 0.1*, *pool_mode: str = 'sum'*, ***kwargs*)

Model for Graph Property Prediction.

This model proceeds as follows:

- **Update node representations in graphs with a variant of GAT, where a**
  linear additive form of attention is applied. Attention Weights are derived by concatenating the node and edge features for each bond.
- Update node representations with multiple rounds of message passing.
- For each layer has, residual connections with its previous layer.
- **The final molecular representation is computed by combining the representations**
  of all nodes in the molecule.
- Perform the final prediction using a linear layer

Examples

```
>>> import deepchem as dc
>>> from deepchem.models import PagtnModel
>>> # preparing dataset
>>> smiles = ["C1CCC1", "CCC"]
>>> labels = [0., 1.]
>>> featurizer = dc.feat.PagtnMolGraphFeaturizer(max_length=5)
>>> X = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(X=X, y=labels)
>>> # training model
>>> model = PagtnModel(mode='classification', n_tasks=1,
...                    batch_size=16, learning_rate=0.001)
>>> loss = model.fit(dataset, nb_epoch=5)
```

References

Notes

This class requires DGL (https://github.com/dmlc/dgl) and DGL-LifeSci (https://github.com/awslabs/dgl-lifesci) to be installed.

`__init__`(*n_tasks: int*, *number_atom_features: int = 94*, *number_bond_features: int = 42*, *mode: str = 'regression'*, *n_classes: int = 2*, *output_node_features: int = 256*, *hidden_features: int = 32*, *num_layers: int = 5*, *num_heads: int = 1*, *dropout: float = 0.1*, *pool_mode: str = 'sum'*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.
- **number_atom_features** (*int*) – Size for the input node features. Default to 94.
- **number_bond_features** (*int*) – Size for the input edge features. Default to 42.
- **mode** (*str*) – The model type, 'classification' or 'regression'. Default to 'regression'.
- **n_classes** (*int*) – The number of classes to predict per task (only used when `mode` is 'classification'). Default to 2.
- **output_node_features** (*int*) – Size for the output node features in PAGTN layers. Default to 256.
- **hidden_features** (*int*) – Size for the hidden node features in PAGTN layers. Default to 32.
- **num_layers** (*int*) – Number of graph neural network layers, i.e. number of rounds of message passing. Default to 2.
- **num_heads** (*int*) – Number of attention heads. Default to 1.
- **dropout** (*float*) – Dropout probability. Default to 0.1
- **pool_mode** (*'max' or 'mean' or 'sum'*) – Whether to compute elementwise maximum, mean or sum of the node representations.
- **kwargs** – This can include any keyword argument of TorchModel.

3.18.12 AtomConvModel

class AtomConvModel(*n_tasks: int*, *frag1_num_atoms: int = 70*, *frag2_num_atoms: int = 634*, *complex_num_atoms: int = 701*, *max_num_neighbors: int = 12*, *batch_size: int = 24*, *atom_types: Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0]*, *radial: Sequence[Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]]*, *layer_sizes=[100]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = ['relu']*, *residual: bool = False*, *learning_rate=0.001*, ***kwargs*)

An Atomic Convolutional Neural Network (ACNN) for energy score prediction.

The network follows the design of a graph convolutional network but in this case the graph is represented as a 3D structure of the molecule. The objective of this model is to train models and predict energetic state starting from the spatial geometry of the model [1].

References

Examples

```
>>> from deepchem.models.torch_models import AtomConvModel
>>> from deepchem.data import NumpyDataset
>>> frag1_num_atoms = 100 # atoms for ligand
>>> frag2_num_atoms = 1200 # atoms for protein
>>> complex_num_atoms = frag1_num_atoms + frag2_num_atoms
>>> batch_size = 1
>>> # Initialize the model
>>> atomic_convnet = AtomConvModel(n_tasks=1,
...                                batch_size=batch_size,
...                                layer_sizes=[
...                                    10,
...                                ],
...                                frag1_num_atoms=frag1_num_atoms,
...                                frag2_num_atoms=frag2_num_atoms,
...                                complex_num_atoms=complex_num_atoms)
>>> # Creates a set of dummy features that contain the coordinate and
>>> # neighbor-list features required by the AtomicConvModel.
>>> # Preparing the dataset
>>> features = []
>>> frag1_coords = np.random.rand(frag1_num_atoms, 3)
>>> frag1_nbr_list = {i: [] for i in range(frag1_num_atoms)}
>>> frag1_z = np.random.randint(10, size=(frag1_num_atoms))
>>> frag2_coords = np.random.rand(frag2_num_atoms, 3)
>>> frag2_nbr_list = {i: [] for i in range(frag2_num_atoms)}
>>> frag2_z = np.random.randint(10, size=(frag2_num_atoms))
>>> system_coords = np.random.rand(complex_num_atoms, 3)
>>> system_nbr_list = {i: [] for i in range(complex_num_atoms)}
>>> system_z = np.random.randint(10, size=(complex_num_atoms))
>>> features.append((frag1_coords, frag1_nbr_list, frag1_z, frag2_coords, frag2_nbr_
↪list, frag2_z, system_coords, system_nbr_list, system_z))
```

(continues on next page)

(continued from previous page)

```
>>> features = np.asarray(features, dtype=object)
>>> labels = np.zeros(batch_size)
>>> train = NumpyDataset(features, labels)
>>> _ = atomic_convnet.fit(train, nb_epoch=1)
>>> preds = atomic_convnet.predict(train)
```

`__init__`(*n_tasks: int*, *frag1_num_atoms: int = 70*, *frag2_num_atoms: int = 634*, *complex_num_atoms: int = 701*, *max_num_neighbors: int = 12*, *batch_size: int = 24*, *atom_types: Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0]*, *radial: Sequence[Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]]*, *layer_sizes=[100]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = ['relu']*, *residual: bool = False*, *learning_rate=0.001*, ***kwargs*) → None

TorchModel wrapper for ACNN

Parameters

- **`n_tasks`** (*`int`*) – number of tasks
- **`frag1_num_atoms`** (*`int`*) – Number of atoms in first fragment.
- **`frag2_num_atoms`** (*`int`*) – Number of atoms in second fragment.
- **`complex_num_atoms`** (*`int`*) – Number of atoms in complex.
- **`max_num_neighbors`** (*`int`*) – Maximum number of neighbors possible for an atom. Recall neighbors are spatial neighbors.
- **`batch_size`** (*`int`*) – Size of the batch.
- **`atom_types`** (*`list`*) – List of atoms recognized by model. Atoms are indicated by their nuclear numbers.
- **`radial`** (*`list`*) – Radial parameters used in the atomic convolution transformation.
- **`layer_sizes`** (*`list`*) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **`weight_init_stddevs`** (*`list or float`*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`bias_init_consts`** (*`list or float`*) – the value to initialize the biases in each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`dropouts`** (*`list or float`*) – the dropout probability to use for each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`activation_fns`** (*`list or object`*) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`residual`** (*`bool`*) – Whether to use residual connections.
- **`learning_rate`** (*`float`*) – the learning rate to use for fitting.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Convert a dataset into the tensors needed for learning.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to convert
- **epochs** (*int, optional (Default 1)*) – Number of times to walk over *dataset*
- **mode** (*str, optional (Default 'fit')*) – Ignored in this implementation.
- **deterministic** (*bool, optional (Default True)*) – Whether the dataset should be walked in a deterministic fashion
- **pad_batches** (*bool, optional (Default True)*) – If true, each returned batch will have size *self.batch_size*.

Return type

Iterator which walks over the batches

3.18.13 MPNNModel

Note that this is an alternative implementation for MPNN and currently you can only import it from `deepchem.models.torch_models`.

class MPNNModel(*n_tasks: int*, *node_out_feats: int = 64*, *edge_hidden_feats: int = 128*, *num_step_message_passing: int = 3*, *num_step_set2set: int = 6*, *num_layer_set2set: int = 3*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *number_bond_features: int = 11*, *n_classes: int = 2*, *self_loop: bool = False*, ***kwargs*)

Model for graph property prediction

This model proceeds as follows:

- **Combine latest node representations and edge features in updating node representations,**
  which involves multiple rounds of message passing
- **For each graph, compute its representation by combining the representations**
  of all nodes in it, which involves a Set2Set layer.
- Perform the final prediction using an MLP

Examples

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models import MPNNModel
>>> # preparing dataset
>>> smiles = ["C1CCC1", "CCC"]
>>> labels = [0., 1.]
>>> featurizer = dc.feat.MolGraphConvFeaturizer(use_edges=True)
>>> X = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(X=X, y=labels)
>>> # training model
>>> model = MPNNModel(mode='classification', n_tasks=1,
...                  batch_size=16, learning_rate=0.001)
>>> loss =  model.fit(dataset, nb_epoch=5)
```

References

Notes

This class requires DGL (https://github.com/dmlc/dgl) and DGL-LifeSci (https://github.com/awslabs/dgl-lifesci) to be installed.

The featurizer used with MPNNModel must produce a GraphData object which should have both 'edge' and 'node' features.

`__init__`(*n_tasks: int*, *node_out_feats: int = 64*, *edge_hidden_feats: int = 128*, *num_step_message_passing: int = 3*, *num_step_set2set: int = 6*, *num_layer_set2set: int = 3*, *mode: str = 'regression'*, *number_atom_features: int = 30*, *number_bond_features: int = 11*, *n_classes: int = 2*, *self_loop: bool = False*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.
- **node_out_feats** (*int*) – The length of the final node representation vectors. Default to 64.
- **edge_hidden_feats** (*int*) – The length of the hidden edge representation vectors. Default to 128.
- **num_step_message_passing** (*int*) – The number of rounds of message passing. Default to 3.
- **num_step_set2set** (*int*) – The number of set2set steps. Default to 6.
- **num_layer_set2set** (*int*) – The number of set2set layers. Default to 3.
- **mode** (*str*) – The model type, 'classification' or 'regression'. Default to 'regression'.
- **number_atom_features** (*int*) – The length of the initial atom feature vectors. Default to 30.
- **number_bond_features** (*int*) – The length of the initial bond feature vectors. Default to 11.
- **n_classes** (*int*) – The number of classes to predict per task (only used when `mode` is 'classification'). Default to 2.
- **self_loop** (*bool*) – Whether to add self loops for the nodes, i.e. edges from nodes to themselves. Generally, an MPNNModel does not require self loops. Default to False.
- **kwargs** – This can include any keyword argument of TorchModel.

3.18.14 InfoGraphModel

`class` **`InfoGraphModel`**(*num_features*, *embedding_dim*, *num_gc_layers=5*, *prior=True*, *gamma=0.1*, *measure='JSD'*, *average_loss=True*, *task='pretraining'*, *n_tasks: int | None = None*, *n_classes: int | None = None*, ***kwargs*)

InfoGraphMode

InfoGraphModel is a model which learn graph-level representation via unsupervised learning. To this end, the model aims to maximize the mutual information between the representations of entire graphs and the representations of substructures of different granularity (eg. nodes, edges, triangles)

The unsupervised training of InfoGraph involves two encoders: one that encodes the entire graph and another that encodes substructures of different sizes. The mutual information between the two encoder outputs is maximized using a contrastive loss function. The model randomly samples pairs of graphs and substructures, and then maximizes their mutual information by minimizing their distance in a learned embedding space.

This can be used for downstream tasks such as graph classification and molecular property prediction.It is implemented as a ModularTorchModel in order to facilitate transfer learning.

References

1. Sun, F.-Y., Hoffmann, J., Verma, V. & Tang, J. InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation Learning via Mutual Information Maximization. Preprint at http://arxiv.org/abs/1908.01000 (2020).

Parameters

- **num_features** (*int*) – Number of node features for each input
- **edge_features** (*int*) – Number of edge features for each input
- **embedding_dim** (*int*) – Dimension of the embedding
- **num_gc_layers** (*int*) – Number of graph convolutional layers
- **prior** (*bool*) – Whether to use a prior expectation in the loss term
- **gamma** (*float*) – Weight of the prior expectation in the loss term
- **measure** (*str*) – The divergence measure to use for the unsupervised loss. Options are 'GAN', 'JSD', 'KL', 'RKL', 'X2', 'DV', 'H2', or 'W1'.
- **average_loss** (*bool*) – Whether to average the loss over the batch
- **n_classes** (*int*) – Number of classses

Example

```
>>> from deepchem.models.torch_models.infograph import InfoGraphModel
>>> from deepchem.feat import MolGraphConvFeaturizer
>>> from deepchem.data import NumpyDataset
>>> import torch
>>> import numpy as np
>>> import tempfile
>>> tempdir = tempfile.TemporaryDirectory()
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> featurizer = MolGraphConvFeaturizer(use_edges=True)
>>> X = featurizer.featurize(smiles)
>>> y = torch.randint(0, 2, size=(2, 1)).float()
>>> w = torch.ones(size=(2, 1)).float()
>>> dataset = NumpyDataset(X, y, w)
>>> num_feat, edge_dim = 30, 11  # num feat and edge dim by molgraph conv featurizer
>>> pretrain_model = InfoGraphModel(num_feat, edge_dim, num_gc_layers=1, task=
↪'pretraining', model_dir=tempdir.name)
>>> pretraining_loss = pretrain_model.fit(dataset, nb_epoch=1)
>>> pretrain_model.save_checkpoint()
>>> finetune_model = InfoGraphModel(num_feat, edge_dim, num_gc_layers=1, task=
↪'regression', n_tasks=1, model_dir=tempdir.name)
>>> finetune_model.restore(components=['encoder'])
```

(continues on next page)

(continued from previous page)

```
>>> finetuning_loss = finetune_model.fit(dataset)
>>>
>>> # classification example
>>> n_classes, n_tasks = 2, 1
>>> classification_model = InfoGraphModel(num_feat, edge_dim, num_gc_layers=1, task=
↪'classification', n_tasks=1, n_classes=2)
>>> y = np.random.randint(n_classes, size=(len(smiles), n_tasks)).astype(np.float64)
>>> dataset = NumpyDataset(X, y, w)
>>> loss = classification_model.fit(dataset, nb_epoch=1)
```

`__init__`(*num_features*, *embedding_dim*, *num_gc_layers=5*, *prior=True*, *gamma=0.1*, *measure='JSD'*, *average_loss=True*, *task='pretraining'*, *n_tasks: int | None = None*, *n_classes: int | None = None*, ***kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

`build_components`() → dict

Build the components of the model. InfoGraph is an unsupervised molecular graph representation learning model. It consists of an encoder, a local discriminator, a global discriminator, and a prior discriminator.

The unsupervised loss is calculated by the mutual information in embedding representations at all layers.

Components list, type and description:

encoder: GINEncoder, graph convolutional encoder

local_d: MultilayerPerceptron, local discriminator

global_d: MultilayerPerceptron, global discriminator

prior_d: MultilayerPerceptron, prior discriminator fc1: MultilayerPerceptron, dense layer used during finetuning fc2: MultilayerPerceptron, dense layer used during finetuning

`build_model`() → Module

Builds the final model from the components.

`loss_func`(*inputs*, *labels*, *weights*)

Defines the loss function for the model which can access the components using self.components. The loss function should take the inputs, labels, and weights as arguments and return the loss.

`restore`(*components: List[str] | None = None*, *checkpoint: str | None = None*, *model_dir: str | None = None*, *map_location: device | None = None*) → None

Restores the state of a ModularTorchModel from a checkpoint file.

If no checkpoint file is provided, it will use the latest checkpoint found in the model directory. If a list of component names is provided, only the state of those components will be restored.

Parameters

- **components** (*Optional[List[str]]*) – A list of component names to restore. If None, all components will be restored.

- **checkpoint** (*Optional[str]*) – The path to the checkpoint file. If None, the latest checkpoint in the model directory will be used.
- **model_dir** (*Optional[str]*) – The path to the model directory. If None, the model directory used to initialize the model will be used.

3.18.15 InfoGraphStarModel

class InfoGraphStarModel(*num_features*, *edge_features*, *embedding_dim*, *task: Literal['supervised', 'semisupervised'] = 'supervised'*, *mode: Literal['regression', 'classification'] = 'regression'*, *num_classes=2*, *num_tasks=1*, *measure='JSD'*, *average_loss=True*, *num_gc_layers=5*, ***kwargs*)

InfographStar is a semi-supervised graph convolutional network for predicting molecular properties. It aims to maximize the mutual information between the graph-level representation and the representations of substructures of different scales. It does this by producing graph-level encodings and substructure encodings, and then using a discriminator to classify if they are from the same molecule or not.

Supervised training is done by using the graph-level encodings to predict the target property. Semi-supervised training is done by adding a loss term that maximizes the mutual information between the graph-level encodings and the substructure encodings to the supervised loss. These modes can be chosen by setting the training_mode parameter.

To conduct training in unsupervised mode, use InfoGraphModel.

References

Parameters

- **num_features** (*int*) – Number of node features for each input
- **edge_features** (*int*) – Number of edge features for each input
- **embedding_dim** (*int*) – Dimension of the embedding
- **training_mode** (*str*) – The mode to use for training. Options are 'supervised', 'semisupervised'. For unsupervised training, use InfoGraphModel.
- **measure** (*str*) – The divergence measure to use for the unsupervised loss. Options are 'GAN', 'JSD', 'KL', 'RKL', 'X2', 'DV', 'H2', or 'W1'.
- **average_loss** (*bool*) – Whether to average the loss over the batch

Examples

```
>>> from deepchem.models.torch_models import InfoGraphStarModel
>>> from deepchem.feat import MolGraphConvFeaturizer
>>> from deepchem.data import NumpyDataset
>>> import torch
>>> smiles = ["C1CCC1", "C1=CC=CN=C1"]
>>> featurizer = MolGraphConvFeaturizer(use_edges=True)
>>> X = featurizer.featurize(smiles)
>>> y = torch.randint(0, 2, size=(2, 1)).float()
>>> w = torch.ones(size=(2, 1)).float()
>>> ds = NumpyDataset(X, y, w)
>>> num_feat = max([ds.X[i].num_node_features for i in range(len(ds))])
```

(continues on next page)

(continued from previous page)

```
>>> edge_dim = max([ds.X[i].num_edge_features for i in range(len(ds))])
>>> model = InfoGraphStarModel(num_feat, edge_dim, 15, training_mode='semisupervised
↪')
>>> loss = model.fit(ds, nb_epoch=1)
```

`__init__`(*num_features*, *edge_features*, *embedding_dim*, *task: Literal['supervised', 'semisupervised'] = 'supervised'*, *mode: Literal['regression', 'classification'] = 'regression'*, *num_classes=2*, *num_tasks=1*, *measure='JSD'*, *average_loss=True*, *num_gc_layers=5*, ***kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

`build_components`()

Builds the components of the InfoGraphStar model. InfoGraphStar works by maximizing the mutual information between the graph-level representation and the representations of substructures of different scales.

It does this by producing graph-level encodings and substructure encodings, and then using a discriminator to classify if they are from the same molecule or not.

The encoder is a graph convolutional network that produces the graph-level encodings and substructure encodings.

In a supervised training mode, only 1 encoder is used and the encodings are not compared, while in a semi-supvervised training mode they are different in order to prevent negative transfer from the pretraining stage.

The local discriminator is a multilayer perceptron that classifies if the substructure encodings are from the same molecule or not while the global discriminator classifies if the graph-level encodings are from the same molecule or not.

Components list, type and description:

encoder: InfoGraphEncoder

unsup_encoder: InfoGraphEncoder for supervised or GINEncoder for unsupervised training

ff1: MultilayerPerceptron, feedforward network

ff2: MultilayerPerceptron, feedforward network

fc1: torch.nn.Linear, fully connected layer

fc2: torch.nn.Linear, fully connected layer

local_d: MultilayerPerceptron, local discriminator

global_d: MultilayerPerceptron, global discriminator

`build_model`()

Builds the InfoGraph model by unpacking the components dictionary and passing them to the InfoGraph nn.module.

loss_func(*inputs*, *labels*, *weights*)

Defines the loss function for the model which can access the components using self.components. The loss function should take the inputs, labels, and weights as arguments and return the loss.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (`Dataset`) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.16 GNNModular

class GNNModular(*gnn_type: str = 'gin'*, *num_layer: int = 3*, *emb_dim: int = 64*, *num_tasks: int = 1*, *num_classes: int = 2*, *graph_pooling: str = 'mean'*, *dropout: int = 0*, *jump_knowledge: str = 'last'*, *task: str = 'edge_pred'*, *mask_rate: float = 0.1*, *mask_edge: bool = True*, *context_size: int = 1*, *neighborhood_size: int = 3*, *context_mode: str = 'cbow'*, *neg_samples: int = 1*, ***kwargs*)

Modular GNN which allows for easy swapping of GNN layers.

Parameters

- **gnn_type** (*str*) – The type of GNN layer to use. Must be one of "gin", "gcn", "graphsage", or "gat".
- **num_layer** (*int*) – The number of GNN layers to use.
- **emb_dim** (*int*) – The dimensionality of the node embeddings.
- **num_tasks** (*int*) – The number of tasks.
- **graph_pooling** (*str*) – The type of graph pooling to use. Must be one of "sum", "mean", "max", "attention" or "set2set". "sum" may cause issues with positive prediction loss.
- **dropout** (*float, optional (default 0)*) – The dropout probability.
- **jump_knowledge** (*str, optional (default "last")*) – The type of jump knowledge to use. [1] Must be one of "last", "sum", "max", or "concat". "last": Use the node representation from the last GNN layer. "concat": Concatenate the node representations from all GNN layers. This will increase the dimensionality of the node representations by a factor of *num_layer*. "max": Take the element-wise maximum of the node representations from all GNN layers. "sum": Take the element-wise sum of the node representations from all GNN layers. This may cause issues with positive prediction loss.

- **task** (*str, optional (default "regression")*) – The type of task. Unsupervised tasks: edge_pred: Edge prediction. Predicts whether an edge exists between two nodes. mask_nodes: Masking nodes. Predicts the masked node. mask_edges: Masking edges. Predicts the masked edge. infomax: Infomax. Maximizes mutual information between local node representations and a pooled global graph representation. context_pred: Context prediction. Predicts the surrounding context of a node. Supervised tasks: "regression" or "classification".
- **mask_rate** (*float, optional (default 0.1)*) – The rate at which to mask nodes or edges for mask_nodes and mask_edges tasks.
- **mask_edge** (*bool, optional (default True)*) – Whether to also mask connecting edges for mask_nodes tasks.
- **context_size** (*int, optional (default 1)*) – The size of the context to use for context prediction tasks.
- **neighborhood_size** (*int, optional (default 3)*) – The size of the neighborhood to use for context prediction tasks.
- **context_mode** (*str, optional (default "cbow")*) – The context mode to use for context prediction tasks. Must be one of "cbow" or "skipgram".
- **neg_samples** (*int, optional (default 1)*) – The number of negative samples to use for context prediction.

Examples

```
>>> import numpy as np
>>> import deepchem as dc
>>> from deepchem.feat.molecule_featurizers import SNAPFeaturizer
>>> from deepchem.models.torch_models.gnn import GNNModular
>>> featurizer = SNAPFeaturizer()
>>> smiles = ["C1=CC=CC=C1", "C1=CC=CC=C1C=O", "C1=CC=CC=C1C(=O)O"]
>>> features = featurizer.featurize(smiles)
>>> dataset = dc.data.NumpyDataset(features, np.zeros(len(features)))
>>> model = GNNModular(task="edge_pred")
>>> loss = model.fit(dataset, nb_epoch=1)
```

References

__init__(*gnn_type: str = 'gin', num_layer: int = 3, emb_dim: int = 64, num_tasks: int = 1, num_classes: int = 2, graph_pooling: str = 'mean', dropout: int = 0, jump_knowledge: str = 'last', task: str = 'edge_pred', mask_rate: float = 0.1, mask_edge: bool = True, context_size: int = 1, neighborhood_size: int = 3, context_mode: str = 'cbow', neg_samples: int = 1, **kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

build_components()

Builds the components of the GNNModular model. It initializes the encoders, batch normalization layers, pooling layers, and head layers based on the provided configuration. The method returns a dictionary containing the following components:

Components list, type and description:

node_type_embedding: torch.nn.Embedding, an embedding layer for node types.

chirality_embedding: torch.nn.Embedding, an embedding layer for chirality tags.

gconvs: torch_geometric.nn.conv.MessagePassing, a list of graph convolutional layers (encoders) based on the specified GNN type (GIN, GCN, or GAT).

batch_norms: torch.nn.BatchNorm1d, a list of batch normalization layers corresponding to the encoders.

pool: Union[function,torch_geometric.nn.aggr.Aggregation], a pooling layer based on the specified graph pooling type (sum, mean, max, attention, or set2set).

head: nn.Linear, a linear layer for the head of the model.

These components are then used to construct the GNN and GNN_head modules for the GNNModular model.

build_gnn(*num_layer*)

Build graph neural network encoding layers by specifying the number of GNN layers.

Parameters
num_layer (*int*) – The number of GNN layers to be created.

Returns
A tuple containing two ModuleLists: 1. encoders: A ModuleList of GNN layers (currently only GIN is supported). 2. batch_norms: A ModuleList of batch normalization layers corresponding to each GNN layer.

Return type
tuple of (torch.nn.ModuleList, torch.nn.ModuleList)

build_model()

Builds the appropriate model based on the specified task.

For the edge prediction task, the model is simply the GNN module because it is an unsupervised task and does not require a prediction head.

Supervised tasks such as node classification and graph regression require a prediction head, so the model is a sequential module consisting of the GNN module followed by the GNN_head module.

loss_func(*inputs*, *labels*, *weights*)

The loss function executed in the training loop, which is based on the specified task.

masked_node_loss_loader(*inputs*)

Produces the loss between the predicted node features and the true node features for masked nodes. Set mask_edge to True to also predict the edge types for masked edges.

masked_edge_loss_loader(*inputs*)

Produces the loss between the predicted edge types and the true edge types for masked edges.

infomax_loss_loader(*inputs*)

Loss that maximizes mutual information between local node representations and a pooled global graph representation. The positive and negative scores represent the similarity between local node representations and global graph representations of simlar and dissimilar graphs, respectively.

Parameters

inputs (*BatchedGraphData*) – BatchedGraphData object containing the node features, edge indices, and graph indices for the batch of graphs.

context_pred_loss_loader(*inputs*)

Loads the context prediction loss for the given input by taking the batched subgraph and context graphs and computing the context prediction loss for each subgraph and context graph pair.

Parameters

inputs (*tuple*) – A tuple containing the following elements: - substruct_batch (BatchedGraphData): Batched subgraph, or neighborhood, graphs. - s_overlap (List[int]): List of overlapping subgraph node indices between the subgraph and context graphs. - context_graphs (BatchedGraphData): Batched context graphs. - c_overlap (List[int]): List of overlapping context node indices between the subgraph and context graphs. - overlap_size (List[int]): List of the number of overlapping nodes between the subgraph and context graphs.

Returns

context_pred_loss – The context prediction loss

Return type

torch.Tensor

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

This default generator is modified from the default generator in dc.models.tensorgraph.tensor_graph.py to support multitask classification. If the task is classification, the labels y_b are converted to a one-hot encoding and reshaped according to the number of tasks and classes.

3.18.17 InfoMax3DModular

class InfoMax3DModular(*task: Literal['pretraining', 'regression', 'classification']*, *hidden_dim: int = 64*, *target_dim: int = 10*, *aggregators: List[str] = ['mean']*, *readout_aggregators: List[str] = ['mean']*, *scalers: List[str] = ['identity']*, *residual: bool = True*, *node_wise_output_layers: int = 2*, *pairwise_distances: bool = False*, *activation: Callable | str = 'relu'*, *reduce_func: str = 'sum'*, *batch_norm: bool = True*, *batch_norm_momentum: float = 0.1*, *propagation_depth: int = 5*, *dropout: float = 0.0*, *readout_layers: int = 2*, *readout_hidden_dim: int = 1*, *fourier_encodings: int = 4*, *update_net_layers: int = 2*, *message_net_layers: int = 2*, *use_node_features: bool = False*, *posttrans_layers: int = 1*, *pretrans_layers: int = 1*, *n_tasks: int = 1*, *n_classes: bool | None = None*, ***kwargs*)

InfoMax3DModular is a modular torch model that uses a 2D PNA model and a 3D Net3D model to maximize the mutual information between their representations. The 2D model can then be used for downstream tasks without the need for 3D coordinates. This is based off the work in [1].

This class expects data in featurized by the RDKitConformerFeaturizer. This featurizer produces features of the type Array[Array[List[GraphData]]]. The outermost array is the dataset array, the second array is the molecule, the list contains the conformers for that molecule and the GraphData object is the featurized graph for that conformer with node_pos_features holding the 3D coordinates. If you are not using RDKitConformerFeaturizer, your input data features should look like this: Dataset[Molecule[Conformers[GraphData]]].

For pretraining, the original paper used a learning rate of 8e-5 with a batch size of 500. For finetuning on quantum mechanical datasets, a learning rate of 7e-5 with a batch size of 128 was used. For finetuning on non-quantum mechanical datasets, a learning rate of 1e-3 with a batch size of 32 was used in the original implementation.

Parameters

- **task** (*Literal['pretrain', 'regression', 'classification']*) – The task of the model
- **hidden_dim** (*int, optional, default = 64*) – The dimension of the hidden layers.
- **target_dim** (*int, optional, default = 10*) – The dimension of the output layer.
- **aggregators** (*List[str]*) – A list of aggregator functions for the PNA model. Options are 'mean', 'sum', 'min', 'max', 'std', 'var', 'moment3', 'moment4', 'moment5'.
- **readout_aggregators** (*List[str]*) – A list of aggregator functions for the readout layer. Options are 'sum', 'max', 'min', 'mean'.
- **scalers** (*List[str]*) – A list of scaler functions for the PNA model. Options are 'identity', 'amplification', 'attenuation'.
- **residual** (*bool, optional (default=True)*) – Whether to use residual connections in the PNA model.
- **node_wise_output_layers** (*int, optional (default=2)*) – The number of output layers for each node in the Net3D model.
- **pairwise_distances** (*bool, optional (default=False)*) – Whether to use pairwise distances in the PNA model.
- **activation** (*Union[Callable, str], optional (default="relu")*) – The activation function to use in the PNA model.
- **reduce_func** (*str, optional (default='sum')*) – The reduce function to use for aggregating messages in the Net3D model.
- **batch_norm** (*bool, optional (default=True)*) – Whether to use batch normalization in the PNA model.
- **batch_norm_momentum** (*float, optional (default=0.1)*) – The momentum for the batch normalization layers.
- **propagation_depth** (*int, optional (default=5)*) – The number of propagation layers in the PNA and Net3D models.
- **dropout** (*float, optional (default=0.0)*) – The dropout rate for the layers in the PNA and Net3D models.
- **readout_layers** (*int, optional (default=2)*) – The number of readout layers in the PNA and Net3D models.
- **readout_hidden_dim** (*int, optional (default=None)*) – The dimension of the hidden layers in the readout network.
- **fourier_encodings** (*int, optional (default=4)*) – The number of Fourier encodings to use in the Net3D model.
- **update_net_layers** (*int, optional (default=2)*) – The number of update network layers in the Net3D model.
- **message_net_layers** (*int, optional (default=2)*) – The number of message network layers in the Net3D model.

- **use_node_features** (*bool, optional (default=False)*) – Whether to use node features as input in the Net3D model.
- **posttrans_layers** (*int, optional (default=1)*) – The number of post-transformation layers in the PNA model.
- **pretrans_layers** (*int, optional (default=1)*) – The number of pre-transformation layers in the PNA model.
- **kwargs** (*dict*) – Additional keyword arguments.

References

Examples

```
>>> from deepchem.feat.graph_data import BatchGraphData
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import␣
↪RDKitConformerFeaturizer
>>> from deepchem.models.torch_models.gnn3d import InfoMax3DModular
>>> import numpy as np
>>> import deepchem as dc
>>> from deepchem.data.datasets import NumpyDataset
>>> smiles = ["C[C@H](F)Cl", "C[C@@H](F)Cl"]
>>> featurizer = RDKitConformerFeaturizer()
>>> data = featurizer.featurize(smiles)
>>> dataset = NumpyDataset(X=data)
>>> model = InfoMax3DModular(task='pretraining',
...                          hidden_dim=64,
...                          target_dim=10,
...                          aggregators=['max'],
...                          readout_aggregators=['mean'],
...                          scalers=['identity'])
>>> loss = model.fit(dataset, nb_epoch=1)
```

__init__(*task: Literal['pretraining', 'regression', 'classification'], hidden_dim: int = 64, target_dim: int = 10, aggregators: List[str] = ['mean'], readout_aggregators: List[str] = ['mean'], scalers: List[str] = ['identity'], residual: bool = True, node_wise_output_layers: int = 2, pairwise_distances: bool = False, activation: Callable | str = 'relu', reduce_func: str = 'sum', batch_norm: bool = True, batch_norm_momentum: float = 0.1, propagation_depth: int = 5, dropout: float = 0.0, readout_layers: int = 2, readout_hidden_dim: int = 1, fourier_encodings: int = 4, update_net_layers: int = 2, message_net_layers: int = 2, use_node_features: bool = False, posttrans_layers: int = 1, pretrans_layers: int = 1, n_tasks: int = 1, n_classes: bool | None = None, **kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

build_components()

Build the components of the InfoMax3DModular model.

Returns

A dictionary containing the '2d' PNA model and the '3d' Net3D model.

Return type
dict

`build_model()`

Build the InfoMax3DModular model. This is the 2D network which is meant to be used for inference.

Returns
The 2D PNA model component.

Return type
PNA

`loss_func`(*inputs*, *labels*, *weights*)

Compute the loss function for the InfoMax3DModular model.

Parameters

- **inputs** (*dgl.DGLGraph*) – The input graph with node features stored under the key 'x' and edge distances stored under the key 'd'.
- **labels** (*torch.Tensor*) – The ground truth labels.
- **weights** (*torch.Tensor*) – The weights for each sample.

Returns
The computed loss value.

Return type
torch.Tensor

3.18.18 LCNNModel

class LCNNModel(*n_occupancy: int = 3*, *n_neighbor_sites_list: int = 19*, *n_permutation_list: int = 6*, *n_task: int = 1*, *dropout_rate: float = 0.4*, *n_conv: int = 2*, *n_features: int = 44*, *sitewise_n_feature: int = 25*, ***kwargs*)

Lattice Convolutional Neural Network (LCNN). Here is a simple example of code that uses the LCNNModel with Platinum 2d Adsorption dataset.

This model takes arbitrary configurations of Molecules on an adsorbate and predicts their formation energy. These formation energies are found using DFT calculations and LCNNModel is to automate that process. This model defines a crystal graph using the distance between atoms. The crystal graph is an undirected regular graph (equal neighbours) and different permutations of the neighbours are pre-computed using the LCNNFeaturizer. On each node for each permutation, the neighbour nodes are concatenated which are further operated. This model has only a node representation. Please confirm the detail algorithms from [1]_.

Examples

```
>>>
>> import deepchem as dc
>> from pymatgen.core import Structure
>> import numpy as np
>> from deepchem.feat import LCNNFeaturizer
>> from deepchem.molnet import load_Platinum_Adsorption
>> PRIMITIVE_CELL = {
..   "lattice": [[2.818528, 0.0, 0.0],
..               [-1.409264, 2.440917, 0.0],
```

(continues on next page)

(continued from previous page)

```
..              [0.0, 0.0, 25.508255]],
..   "coords": [[0.66667, 0.33333, 0.090221],
..              [0.33333, 0.66667, 0.18043936],
..              [0.0, 0.0, 0.27065772],
..              [0.66667, 0.33333, 0.36087608],
..              [0.33333, 0.66667, 0.45109444],
..              [0.0, 0.0, 0.49656991]],
..   "species": ['H', 'H', 'H', 'H', 'H', 'He'],
..   "site_properties": {'SiteTypes': ['S1', 'S1', 'S1', 'S1', 'S1', 'A1']}
.. }
>> PRIMITIVE_CELL_INF0 = {
..    "cutoff": np.around(6.00),
..    "structure": Structure(**PRIMITIVE_CELL),
..    "aos": ['1', '0', '2'],
..    "pbc": [True, True, False],
..    "ns": 1,
..    "na": 1
.. }
>> tasks, datasets, transformers = load_Platinum_Adsorption(
..    featurizer= LCNNFeaturizer( **PRIMITIVE_CELL_INF0)
.. )
>> train, val, test = datasets
>> model = LCNNModel(mode='regression',
..                   batch_size=8,
..                   learning_rate=0.001)
>> model = LCNN()
>> out = model(lcnn_feat)
>> model.fit(train, nb_epoch=10)
```

References

Notes

This class requires DGL and PyTorch to be installed.

`__init__`(*n_occupancy: int = 3*, *n_neighbor_sites_list: int = 19*, *n_permutation_list: int = 6*, *n_task: int = 1*, *dropout_rate: float = 0.4*, *n_conv: int = 2*, *n_features: int = 44*, *sitewise_n_feature: int = 25*, ***kwargs*)

This class accepts all the keyword arguments from TorchModel.

Parameters

- **n_occupancy** (*int, default 3*) – number of possible occupancy.
- **n_neighbor_sites_list** (*int, default 19*) – Number of neighbors of each site.
- **n_permutation** (*int, default 6*) – Diffrent permutations taken along diffrent directions.
- **n_task** (*int, default 1*) – Number of tasks.
- **dropout_rate** (*float, default 0.4*) – p value for dropout between 0.0 to 1.0
- **nconv** (*int, default 2*) – number of convolutions performed.
- **n_feature** (*int, default 44*) – number of feature for each site.

- **sitewise_n_feature** (*int, default 25*) – number of features for atoms for site-wise activation.
- **kwargs** (*Dict*) – This class accepts all the keyword arguments from TorchModel.

3.18.19 MEGNetModel

class MEGNetModel(*n_node_features: int = 32*, *n_edge_features: int = 32*, *n_global_features: int = 32*, *n_blocks: int = 1*, *is_undirected: bool = True*, *residual_connection: bool = True*, *mode: str = 'regression'*, *n_classes: int = 2*, *n_tasks: int = 1*, ***kwargs*)

MatErials Graph Network for Molecules and Crystals

MatErials Graph Network [1]_ are Graph Networks [2]_ which are used for property prediction in molecules and crystals. The model implements multiple layers of Graph Network as MEGNetBlocks and then combines the node properties and edge properties of all nodes and edges via a Set2Set layer. The combines information is used with the global features of the material/molecule for property prediction tasks.

Example

```
>>> import deepchem as dc
>>> from deepchem.models import MEGNetModel
>>> from deepchem.utils.fake_data_generator import FakeGraphGenerator as FGG
>>> graphs = FGG(global_features=4, num_classes=10).sample(n_graphs=20)
>>> model = dc.models.MEGNetModel(n_node_features=5, n_edge_features=3, n_global_
↪features=4, n_blocks=3, is_undirected=True, residual_connection=True, mode=
↪'classification', n_classes=10, batch_size=16)
>>> training_loss = model.fit(graphs)
```

References

Note: The model requires PyTorch-Geometric to be installed.

__init__(*n_node_features: int = 32*, *n_edge_features: int = 32*, *n_global_features: int = 32*, *n_blocks: int = 1*, *is_undirected: bool = True*, *residual_connection: bool = True*, *mode: str = 'regression'*, *n_classes: int = 2*, *n_tasks: int = 1*, ***kwargs*)

Parameters

- **n_node_features** (*int*) – Number of features in a node
- **n_edge_features** (*int*) – Number of features in a edge
- **n_global_features** (*int*) – Number of global features
- **n_blocks** (*int*) – Number of GraphNetworks block to use in update
- **is_undirected** (*bool, optional (default True)*) – True when the model is used on undirected graphs otherwise false
- **residual_connection** (*bool, optional (default True)*) – If True, the layer uses a residual connection during training
- **n_tasks** (*int, default 1*) – The number of tasks

- **mode** (*str, default 'regression'*) – The model type - classification or regression
- **n_classes** (*int, default 2*) – The number of classes to predict (used only in classification mode).
- **kwargs** (*Dict*) – kwargs supported by TorchModel

3.18.20 MATModel

class MATModel(*dist_kernel: str = 'softmax', n_encoders=8, lambda_attention: float = 0.33, lambda_distance: float = 0.33, h: int = 16, sa_hsize: int = 1024, sa_dropout_p: float = 0.0, output_bias: bool = True, d_input: int = 1024, d_hidden: int = 1024, d_output: int = 1024, activation: str = 'leakyrelu', n_layers: int = 1, ff_dropout_p: float = 0.0, encoder_hsize: int = 1024, encoder_dropout_p: float = 0.0, embed_input_hsize: int = 36, embed_dropout_p: float = 0.0, gen_aggregation_type: str = 'mean', gen_dropout_p: float = 0.0, gen_n_layers: int = 1, gen_attn_hidden: int = 128, gen_attn_out: int = 4, gen_d_output: int = 1, **kwargs*)

Molecular Attention Transformer.

This class implements the Molecular Attention Transformer [1]_. The MATFeaturizer (deepchem.feat.MATFeaturizer) is intended to work with this class. The model takes a batch of MATEncodings (from MATFeaturizer) as input, and returns an array of size Nx1, where N is the number of molecules in the batch. Each molecule is broken down into its Node Features matrix, adjacency matrix and distance matrix. A mask tensor is calculated for the batch. All of this goes as input to the MATEmbedding, MATEncoder and MATGenerator layers, which are defined in deepchem.models.torch_models.layers.py

Currently, MATModel is intended to be a regression model for the freesolv dataset.

References

Examples

```
>>> import deepchem as dc
>>> import pandas as pd
>>> smiles = ['CC', 'CCC',  'CCCC', 'CCCCC', 'CCCCCCC']
>>> vals = [1.35, 6.72, 5.67, 1.23, 1.76]
>>> df = pd.DataFrame(list(zip(smiles, vals)), columns = ['smiles', 'y'])
>>> loader = dc.data.CSVLoader(tasks=['y'], feature_field='smiles', featurizer=dc.
↪feat.MATFeaturizer())
>>> df.to_csv('test.csv')
>>> dataset = loader.create_dataset('test.csv')
>>> model = dc.models.torch_models.MATModel(batch_size = 2)
>>> out = model.fit(dataset, nb_epoch = 1)
```

__init__(*dist_kernel: str = 'softmax', n_encoders=8, lambda_attention: float = 0.33, lambda_distance: float = 0.33, h: int = 16, sa_hsize: int = 1024, sa_dropout_p: float = 0.0, output_bias: bool = True, d_input: int = 1024, d_hidden: int = 1024, d_output: int = 1024, activation: str = 'leakyrelu', n_layers: int = 1, ff_dropout_p: float = 0.0, encoder_hsize: int = 1024, encoder_dropout_p: float = 0.0, embed_input_hsize: int = 36, embed_dropout_p: float = 0.0, gen_aggregation_type: str = 'mean', gen_dropout_p: float = 0.0, gen_n_layers: int = 1, gen_attn_hidden: int = 128, gen_attn_out: int = 4, gen_d_output: int = 1, **kwargs*)

The wrapper class for the Molecular Attention Transformer.

Since we are using a custom data class as input (MATEncoding), we have overriden the default_generator function from DiskDataset and customized it to work with a batch of MATEncoding classes.

Parameters

- **dist_kernel** (*str*) – Kernel activation to be used. Can be either 'softmax' for softmax or 'exp' for exponential, for the self-attention layer.
- **n_encoders** (*int*) – Number of encoder layers in the encoder block.
- **lambda_attention** (*float*) – Constant to be multiplied with the attention matrix in the self-attention layer.
- **lambda_distance** (*float*) – Constant to be multiplied with the distance matrix in the self-attention layer.
- **h** (*int*) – Number of attention heads for the self-attention layer.
- **sa_hsize** (*int*) – Size of dense layer in the self-attention layer.
- **sa_dropout_p** (*float*) – Dropout probability for the self-attention layer.
- **output_bias** (*bool*) – If True, dense layers will use bias vectors in the self-attention layer.
- **d_input** (*int*) – Size of input layer in the feed-forward layer.
- **d_hidden** (*int*) – Size of hidden layer in the feed-forward layer. Will also be used as d_output for the MATEmbedding layer.
- **d_output** (*int*) – Size of output layer in the feed-forward layer.
- **activation** (*str*) – Activation function to be used in the feed-forward layer. Can choose between 'relu' for ReLU, 'leakyrelu' for LeakyReLU, 'prelu' for PReLU, 'tanh' for TanH, 'selu' for SELU, 'elu' for ELU and 'linear' for linear activation.
- **n_layers** (*int*) – Number of layers in the feed-forward layer.
- **ff_dropout_p** (*float*) – Dropout probability in the feeed-forward layer.
- **encoder_hsize** (*int*) – Size of Dense layer for the encoder itself.
- **encoder_dropout_p** (*float*) – Dropout probability for connections in the encoder layer.
- **embed_input_hsize** (*int*) – Size of input layer for the MATEmbedding layer.
- **embed_dropout_p** (*float*) – Dropout probability for the MATEmbedding layer.
- **gen_aggregation_type** (*str*) – Type of aggregation to be used. Can be 'grover', 'mean' or 'contextual'.
- **gen_dropout_p** (*float*) – Dropout probability for the MATGenerator layer.
- **gen_n_layers** (*int*) – Number of layers in MATGenerator.
- **gen_attn_hidden** (*int*) – Size of hidden attention layer in the MATGenerator layer.
- **gen_attn_out** (*int*) – Size of output attention layer in the MATGenerator layer.
- **gen_d_output** (*int*) – Size of output layer in the MATGenerator layer.

pad_array(*array: ndarray*, *shape: Any*) → ndarray

Pads an array to the desired shape.

Parameters

- **array** (*np.ndarray*) –
- **padded.** (*Array to be*) –
- **shape** (*int or Tuple*) –
- **to.** (*Shape the array is padded*) –

Returns

- **array** (*np.ndarray*)
- *Array padded to input shape.*

`pad_sequence`(*sequence: ndarray*) → ndarray

Pads a given sequence using the pad_array function.

Parameters

- **sequence** (`np.ndarray`) –
- **sequence.** (`Arrays in this sequence are padded to the largest shape in the`) –

Returns

- **array** (*np.ndarray*)
- *Sequence with padded arrays.*

`default_generator`(*dataset*, *epochs=1*, *mode='fit'*, *deterministic=True*, *pad_batches=True*, ***kwargs*)

Create a generator that iterates batches for a dataset.

Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (`Dataset`) – the data to iterate
- **epochs** (`int`) – the number of times to iterate over the full dataset
- **mode** (`str`) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (`bool`) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (`bool`) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.21 NormalizingFlowModel

`class NormalizingFlow`(*transform: Sequence*, *base_distribution*, *dim: int*)

Normalizing flows are widley used to perform generative models. This algorithm gives advantages over variational autoencoders (VAE) because of ease in sampling by applying invertible transformations (Frey, Gadepally, & Ramsundar, 2022).

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.layers import Affine
>>> from deepchem.models.torch_models.normalizing_flows_pytorch import␣
↪NormalizingFlow
>>> import torch
>>> from torch.distributions import MultivariateNormal
>>> # initialize the transformation layer's parameters
>>> dim = 2
>>> samples = 96
>>> transforms = [Affine(dim)]
>>> distribution = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> # initialize normalizing flow model
>>> model = NormalizingFlow(transforms, distribution, dim)
>>> # evaluate the log_prob when applying the transformation layers
>>> input = distribution.sample(torch.Size((samples, dim)))
>>> len(model.log_prob(input))
96
>>> # evaluates the the sampling method and its log_prob
>>> len(model.sample(samples))
2
```

3.18.22 DMPNNModel

`class DMPNNModel`(*mode: str = 'regression', n_classes: int = 3, n_tasks: int = 1, batch_size: int = 1, global_features_size: int = 0, use_default_fdim: bool = True, atom_fdim: int = 133, bond_fdim: int = 14, enc_hidden: int = 300, depth: int = 3, bias: bool = False, enc_activation: str = 'relu', enc_dropout_p: float = 0.0, aggregation: str = 'mean', aggregation_norm: int | float = 100, ffn_hidden: int = 300, ffn_activation: str = 'relu', ffn_layers: int = 3, ffn_dropout_p: float = 0.0, ffn_dropout_at_input_no_act: bool = True, **kwargs*)

Directed Message Passing Neural Network

This class implements the Directed Message Passing Neural Network (D-MPNN) [1]_.

The DMPNN model has 2 phases, message-passing phase and read-out phase.

- The goal of the message-passing phase is to generate 'hidden states of all the atoms in the molecule' using encoders.
- Next in read-out phase, the features are passed into feed-forward neural network to get the task-based prediction.

For additional information:

- Mapper class
- Encoder layer class
- Feed-Forward class

Example

```
>>> import deepchem as dc
>>> import os
>>> model_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
>>> input_file = os.path.join(model_dir, 'tests/assets/freesolv_sample_5.csv')
>>> loader = dc.data.CSVLoader(tasks=['y'], feature_field='smiles', featurizer=dc.
↪feat.DMPNNFeaturizer())
>>> dataset = loader.create_dataset(input_file)
>>> model = DMPNNModel()
>>> out = model.fit(dataset, nb_epoch=1)
```

References

`__init__`(*mode: str = 'regression', n_classes: int = 3, n_tasks: int = 1, batch_size: int = 1, global_features_size: int = 0, use_default_fdim: bool = True, atom_fdim: int = 133, bond_fdim: int = 14, enc_hidden: int = 300, depth: int = 3, bias: bool = False, enc_activation: str = 'relu', enc_dropout_p: float = 0.0, aggregation: str = 'mean', aggregation_norm: int | float = 100, ffn_hidden: int = 300, ffn_activation: str = 'relu', ffn_layers: int = 3, ffn_dropout_p: float = 0.0, ffn_dropout_at_input_no_act: bool = True, **kwargs*)

Initialize the DMPNNModel class.

Parameters

- **`mode`** (*`str, default 'regression'`*) – The model type - classification or regression.
- **`n_classes`** (*`int, default 3`*) – The number of classes to predict (used only in classification mode).
- **`n_tasks`** (*`int, default 1`*) – The number of tasks.
- **`batch_size`** (*`int, default 1`*) – The number of datapoints in a batch.
- **`global_features_size`** (*`int, default 0`*) – Size of the global features vector, based on the global featurizers used during featurization.
- **`use_default_fdim`** (*`bool`*) – If *True*, self.atom_fdim and self.bond_fdim are initialized using values from the GraphConvConstants class. If *False*, self.atom_fdim and self.bond_fdim are initialized from the values provided.
- **`atom_fdim`** (*`int`*) – Dimension of atom feature vector.
- **`bond_fdim`** (*`int`*) – Dimension of bond feature vector.
- **`enc_hidden`** (*`int`*) – Size of hidden layer in the encoder layer.
- **`depth`** (*`int`*) – No of message passing steps.
- **`bias`** (*`bool`*) – If *True*, dense layers will use bias vectors.
- **`enc_activation`** (*`str`*) – Activation function to be used in the encoder layer. Can choose between 'relu' for ReLU, 'leakyrelu' for LeakyReLU, 'prelu' for PReLU, 'tanh' for TanH, 'selu' for SELU, and 'elu' for ELU.
- **`enc_dropout_p`** (*`float`*) – Dropout probability for the encoder layer.
- **`aggregation`** (*`str`*) – Aggregation type to be used in the encoder layer. Can choose between 'mean', 'sum', and 'norm'.

- **aggregation_norm** (*Union[int, float]*) – Value required if *aggregation* type is 'norm'.
- **ffn_hidden** (*int*) – Size of hidden layer in the feed-forward network layer.
- **ffn_activation** (*str*) – Activation function to be used in feed-forward network layer. Can choose between 'relu' for ReLU, 'leakyrelu' for LeakyReLU, 'prelu' for PReLU, 'tanh' for TanH, 'selu' for SELU, and 'elu' for ELU.
- **ffn_layers** (*int*) – Number of layers in the feed-forward network layer.
- **ffn_dropout_p** (*float*) – Dropout probability for the feed-forward network layer.
- **ffn_dropout_at_input_no_act** (*bool*) – If true, dropout is applied on the input tensor. For single layer, it is not passed to an activation function.
- **kwargs** (*Dict*) – kwargs supported by TorchModel

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = False*, ***kwargs*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset.

Overrides the existing `default_generator` method to customize how model inputs are generated from the data.

Here, the `_MapperDMPNN` helper class is used, for each molecule in a batch, to get required input parameters:

- atom_features
- f_ini_atoms_bonds
- atom_to_incoming_bonds
- mapping
- global_features

Then data from each molecule is converted to a `_ModData` object and stored as list of graphs. The graphs are modified such that all tensors have same size in 0th dimension. (important requirement for batching)

Parameters

- **dataset** (`Dataset`) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*
- *Here, [inputs] is list of graphs.*

3.18.23 GroverModel

class GroverModel(*node_fdim: int*, *edge_fdim: int*, *hidden_size: int*, *self_attention=False*, *features_only=False*, *atom_vocab:* GroverAtomVocabularyBuilder *| None = None*, *bond_vocab: GroverBondVocabularyBuilder | None = None*, *functional_group_size: int | None = 85*, *features_dim: int = 128*, *dropout: float = 0.2*, *activation: str = 'relu'*, *task: str = 'pretraining'*, *ffn_num_layers: int = 1*, *ffn_hidden_size: int = 64*, *attn_out_size: int = 16*, *num_attn_heads: int = 4*, *depth: int = 1*, *mode: str | None = None*, *model_dir=None*, *n_tasks: int = 1*, *n_classes: int | None = None*, ***kwargs*)

GROVER model

The GROVER model employs a self-supervised message passing transformer architecutre for learning molecular representation. The pretraining task can learn rich structural and semantic information of molecules from unlabelled molecular data, which can be leveraged by finetuning for downstream applications. To this end, GROVER integrates message passing networks into a transformer style architecture.

Parameters

- **node_fdim** (*int*) – the dimension of additional feature for node/atom.
- **edge_fdim** (*int*) – the dimension of additional feature for edge/bond.
- **atom_vocab** (GroverAtomVocabularyBuilder) – Grover atom vocabulary builder required during pretraining.
- **bond_vocab** (*GroverBondVocabularyBuilder*) – Grover bond vocabulary builder required during pretraining.
- **hidden_size** (*int*) – Size of hidden layers
- **features_only** (*bool*) – Uses only additional features in the feed-forward network, no graph network
- **self_attention** (*bool, default False*) – When set to True, a self-attention layer is used during graph readout operation.
- **functional_group_size** (*int (default: 85)*) – Size of functional group used in grover.
- **features_dim** (*int*) – Size of additional molecular features, like fingerprints.
- **ffn_num_layers** (*int (default: 1)*) – Number of linear layers to use for feature extraction from embeddings
- **ffn_hidden_size** (*int (default: 64)*) – Hidden size of feed forward network
- **attn_out_size** (*int (default: 16)*) – Size of attention heads
- **num_attn_heads** (*int (default: 4)*) – Number of attention heads
- **task** (*str (pretraining or finetuning)*) – Pretraining or finetuning tasks.
- **mode** (*str (classification or regression)*) – Training mode (used only for finetuning)
- **n_tasks** (*int, optional (default: 1)*) – Number of tasks
- **n_classes** (*int, optiona (default: 2)*) – Number of target classes in classification mode
- **model_dir** (*str*) – Directory to save model checkpoints
- **dropout** (*float, optional (default: 0.2)*) – dropout value

- **activation** (*str, optional (default: 'relu')*) – supported activation function
- **depth** (*int (default: 1)*) – Dynamic message passing depth for use in MPNEncoder

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.grover import GroverModel
>>> from deepchem.feat.vocabulary_builders import (GroverAtomVocabularyBuilder,␣
↪GroverBondVocabularyBuilder)
>>> import pandas as pd
>>> import os
>>> import tempfile
>>> tmpdir = tempfile.mkdtemp()
>>> df = pd.DataFrame({'smiles': ['CC', 'CCC'], 'preds': [0, 0]})
>>> filepath = os.path.join(tmpdir, 'example.csv')
>>> df.to_csv(filepath, index=False)
>>> dataset_path = os.path.join(filepath)
>>> loader = dc.data.CSVLoader(tasks=['preds'], featurizer=dc.feat.
↪DummyFeaturizer(), feature_field=['smiles'])
>>> dataset = loader.create_dataset(filepath)
>>> av = GroverAtomVocabularyBuilder()
>>> av.build(dataset)
>>> bv = GroverBondVocabularyBuilder()
>>> bv.build(dataset)
>>> fg = dc.feat.CircularFingerprint()
>>> loader2 = dc.data.CSVLoader(tasks=['preds'], featurizer=dc.feat.
↪GroverFeaturizer(features_generator=fg), feature_field='smiles')
>>> graph_data = loader2.create_dataset(filepath)
>>> model = GroverModel(node_fdim=151, edge_fdim=165, atom_vocab=av, bond_vocab=bv,␣
↪features_dim=2048, hidden_size=128, functional_group_size=85, mode='regression',␣
↪task='finetuning', model_dir='gm')
>>> loss = model.fit(graph_data, nb_epoch=1)
```

Reference

__init__(*node_fdim: int*, *edge_fdim: int*, *hidden_size: int*, *self_attention=False*, *features_only=False*, *atom_vocab: GroverAtomVocabularyBuilder | None = None*, *bond_vocab: GroverBondVocabularyBuilder | None = None*, *functional_group_size: int | None = 85*, *features_dim: int = 128*, *dropout: float = 0.2*, *activation: str = 'relu'*, *task: str = 'pretraining'*, *ffn_num_layers: int = 1*, *ffn_hidden_size: int = 64*, *attn_out_size: int = 16*, *num_attn_heads: int = 4*, *depth: int = 1*, *mode: str | None = None*, *model_dir=None*, *n_tasks: int = 1*, *n_classes: int | None = None*, ***kwargs*)

Create a ModularTorchModel.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **components** (*dict*) – A dictionary of the components of the model. The keys are the names of the components and the values are the components themselves.

build_components()

Builds components for grover pretraining and finetuning model.

Table 5: Components of pretraining model

Component name	Type	Description
embedding	Graph message passing network	A layer which accepts a molecular graph and produces an embedding for grover pretraining task
atom_vocab_task_atom	Feed forward layer	A layer which accepts an embedding generated from atom hidden states and predicts atom vocabulary for grover pretraining task
atom_vocab_task_bond	Feed forward layer	A layer which accepts an embedding generated from bond hidden states and predicts atom vocabulary for grover pretraining task
bond_vocab_task_atom	Feed forward layer	A layer which accepts an embedding generated from atom hidden states and predicts bond vocabulary for grover pretraining task
bond_vocab_task_bond	Feed forward layer	A layer which accepts an embedding generated from bond hidden states and predicts bond vocabulary for grover pretraining task
functional_group_predictor	Feed forward layer	A layer which accepts an embedding generated from a graph readout and predicts functional group for grover pretraining task

Table 6: Components of finetuning model

Component name	Type	Description
embedding	Graph message passing network	An embedding layer to generate embedding from input molecular graph
readout	Feed forward layer	A readout layer to perform readout atom and bond hidden states
mol_atom_fron	Feed forward layer	A feed forward network which learns representation from atom messages generated via atom hidden states of a molecular graph
mol_atom_fron	Feed forward layer	A feed forward network which learns representation from atom messages generated via bond hidden states of a molecular graph

build_model()

Builds grover pretrain or finetune model based on task

get_loss_func()

Returns loss function based on task

loss_func(*inputs*, *labels*, *weights*)

Returns loss function which performs forward iteration based on task type

static atom_vocab_random_mask(*atom_vocab:* GroverAtomVocabularyBuilder, *smiles: List[str]*) → List[int]

Random masking of atom labels from vocabulary

For every atom in the list of SMILES string, the algorithm fetches the atoms context (vocab label) from the vocabulary provided and returns the vocabulary labels with a random masking (probability of masking = 0.15).

Parameters

- **atom_vocab** (GroverAtomVocabularyBuilder) – atom vocabulary
- **smiles** (*List[str]*) – a list of smiles string

Returns
vocab_label – atom vocab label with random masking

Return type
List[int]

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.grover import GroverModel
>>> from deepchem.feat.vocabulary_builders import GroverAtomVocabularyBuilder
>>> smiles = np.array(['CC', 'CCC'])
>>> dataset = dc.data.NumpyDataset(X=smiles)
>>> atom_vocab = GroverAtomVocabularyBuilder()
>>> atom_vocab.build(dataset)
>>> vocab_labels = GroverModel.atom_vocab_random_mask(atom_vocab, smiles)
```

static bond_vocab_random_mask(*bond_vocab: GroverBondVocabularyBuilder*, *smiles: List[str]*) → List[int]

Random masking of bond labels from bond vocabulary

For every bond in the list of SMILES string, the algorithm fetches the bond context (vocab label) from the vocabulary provided and returns the vocabulary labels with a random masking (probability of masking = 0.15).

Parameters

- **bond_vocab** (*GroverBondVocabularyBuilder*) – bond vocabulary
- **smiles** (*List[str]*) – a list of smiles string

Returns
vocab_label – bond vocab label with random masking

Return type
List[int]

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.grover import GroverModel
>>> from deepchem.feat.vocabulary_builders import GroverBondVocabularyBuilder
>>> smiles = np.array(['CC', 'CCC'])
>>> dataset = dc.data.NumpyDataset(X=smiles)
>>> bond_vocab = GroverBondVocabularyBuilder()
>>> bond_vocab.build(dataset)
>>> vocab_labels = GroverModel.bond_vocab_random_mask(bond_vocab, smiles)
```

restore(*checkpoint: str | None = None*, *model_dir: str | None = None*) → None

Reload the values of all variables from a checkpoint file.

Parameters

- **checkpoint** (*str*) – the path to the checkpoint file to load. If this is None, the most recent checkpoint will be chosen automatically. Call get_checkpoints() to get a list of all available checkpoints.
- **model_dir** (*str, default None*) – Directory to restore checkpoint from. If None, use self.model_dir. If checkpoint is not None, this is ignored.

3.18.24 DTNNModel

class DTNNModel(*n_tasks: int*, *n_embedding: int = 30*, *n_hidden: int = 100*, *n_distance: int = 100*, *distance_min: float = -1*, *distance_max: float = 18*, *output_activation: bool = True*, *mode: str = 'regression'*, *dropout: float = 0.0*, *n_steps: int = 2*, ***kwargs*)

Implements DTNN models for regression.

DTNN is based on the many-body Hamiltonian concept, which is a fundamental principle in quantum mechanics. DTNN recieves a molecule's distance matrix and membership of its atom from its Coulomb Matrix representation. Then, it iteratively refines the representation of each atom by considering its interactions with neighboring atoms. Finally, it predicts the energy of the molecule by summing up the energies of the individual atoms.

This class implements the Deep Tensor Neural Network (DTNN) [1]_.

Examples

```
>>> import os
>>> from deepchem.data import SDFLoader
>>> from deepchem.feat import CoulombMatrix
>>> from deepchem.models.torch_models import DTNNModel
>>> model_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
>>> dataset_file = os.path.join(model_dir, 'tests/assets/qm9_mini.sdf')
>>> TASKS = ["alpha", "homo"]
>>> loader = SDFLoader(tasks=TASKS, featurizer=CoulombMatrix(29), sanitize=True)
>>> data = loader.create_dataset(dataset_file, shard_size=100)
>>> n_tasks = data.y.shape[1]
>>> model = DTNNModel(n_tasks,
...                   n_embedding=20,
...                   n_distance=100,
...                   learning_rate=1.0,
...                   mode="regression")
>>> loss = model.fit(data, nb_epoch=250)
>>> pred = model.predict(data)
```

References

`__init__`(*n_tasks: int*, *n_embedding: int = 30*, *n_hidden: int = 100*, *n_distance: int = 100*, *distance_min: float = -1*, *distance_max: float = 18*, *output_activation: bool = True*, *mode: str = 'regression'*, *dropout: float = 0.0*, *n_steps: int = 2*, ***kwargs*)

Parameters

- **`n_tasks`** (*`int`*) – Number of tasks
- **`n_embedding`** (*`int (default 30)`*) – Number of features per atom.
- **`n_hidden`** (*`int (default 100)`*) – Number of features for each molecule after DTNNStep
- **`n_distance`** (*`int (default 100)`*) – granularity of distance matrix step size will be (distance_max-distance_min)/n_distance
- **`distance_min`** (*`float (default -1)`*) – minimum distance of atom pairs (in Angstrom)
- **`distance_max`** (*`float (default = 18)`*) – maximum distance of atom pairs (in Angstrom)
- **`output_activation`** (*`bool (default True)`*) – determines whether an activation function should be apply to its output.
- **`mode`** (*`str (default "regression")`*) – Only “regression” is currently supported.
- **`dropout`** (*`float (default 0.0)`*) – the dropout probablity to use.
- **`n_steps`** (*`int (default 2)`*) – Number of DTNNStep Layers to use.

`default_generator`(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*)

Create a generator that iterates batches for a dataset. It processes inputs through the _compute_features_on_batch function to calculate required features of input.

Parameters

- **`dataset`** (`Dataset`) – the data to iterate
- **`epochs`** (*`int`*) – the number of times to iterate over the full dataset
- **`mode`** (*`str`*) – allowed values are ‘fit’ (called during training), ‘predict’ (called during prediction), and ‘uncertainty’ (called during uncertainty prediction)
- **`deterministic`** (*`bool`*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **`pad_batches`** (*`bool`*) – whether to pad each batch up to this model’s preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.18.25 SeqToSeqModel

class SeqToSeqModel(*input_tokens: List*, *output_tokens: List*, *max_output_length: int*, *encoder_layers: int = 4*, *decoder_layers: int = 4*, *batch_size: int = 100*, *embedding_dimension: int = 512*, *dropout: float = 0.0*, *reverse_input: bool = True*, *variational: bool = False*, *annealing_start_step: int = 5000*, *annealing_final_step: int = 10000*, ***kwargs*)

Implements sequence to sequence translation models.

The model is based on the description in Sutskever et al., "Sequence to Sequence Learning with Neural Networks" (https://arxiv.org/abs/1409.3215), although this implementation uses GRUs instead of LSTMs. The goal is to take sequences of tokens as input, and translate each one into a different output sequence. The input and output sequences can both be of variable length, and an output sequence need not have the same length as the input sequence it was generated from. For example, these models were originally developed for use in natural language processing. In that context, the input might be a sequence of English words, and the output might be a sequence of French words. The goal would be to train the model to translate sentences from English to French.

The model consists of two parts called the "encoder" and "decoder". Each one consists of a stack of recurrent layers. The job of the encoder is to transform the input sequence into a single, fixed length vector called the "embedding". That vector contains all relevant information from the input sequence. The decoder then transforms the embedding vector into the output sequence.

These models can be used for various purposes. First and most obviously, they can be used for sequence to sequence translation. In any case where you have sequences of tokens, and you want to translate each one into a different sequence, a SeqToSeq model can be trained to perform the translation.

Another possible use case is transforming variable length sequences into fixed length vectors. Many types of models require their inputs to have a fixed shape, which makes it difficult to use them with variable sized inputs (for example, when the input is a molecule, and different molecules have different numbers of atoms). In that case, you can train a SeqToSeq model as an autoencoder, so that it tries to make the output sequence identical to the input one. That forces the embedding vector to contain all information from the original sequence. You can then use the encoder for transforming sequences into fixed length embedding vectors, suitable to use as inputs to other types of models.

Another use case is to train the decoder for use as a generative model. Here again you begin by training the SeqToSeq model as an autoencoder. Once training is complete, you can supply arbitrary embedding vectors, and transform each one into an output sequence. When used in this way, you typically train it as a variational autoencoder. This adds random noise to the encoder, and also adds a constraint term to the loss that forces the embedding vector to have a unit Gaussian distribution. You can then pick random vectors from a Gaussian distribution, and the output sequences should follow the same distribution as the training data.

When training as a variational autoencoder, it is best to use KL cost annealing, as described in https://arxiv.org/abs/1511.06349. The constraint term in the loss is initially set to 0, so the optimizer just tries to minimize the reconstruction loss. Once it has made reasonable progress toward that, the constraint term can be gradually turned back on. The range of steps over which this happens is configurable.

In this class, we establish a sequential model for the Sequence to Sequence (DTNN) **[1]_**.

Examples

```
>>> import torch
>>> from deepchem.models.torch_models.seqtoseq import SeqToSeqModel
>>> data = [
...     ("Cc1cccc(N2CCN(C(=O)C34CC5CC(CC(C5)C3)C4)CC2)c1C",
...      "Cc1cccc(N2CCN(C(=O)C34CC5CC(CC(C5)C3)C4)CC2)c1C"),
...     ("Cn1ccnc1SCC(=O)Nc1ccc(Oc2ccccc2)cc1",
...      "Cn1ccnc1SCC(=O)Nc1ccc(Oc2ccccc2)cc1"),
...     ("COc1cc2c(cc1NC(=O)CN1C(=O)NC3(CCc4ccccc43)C1=O)oc1ccccc12",
...      "COc1cc2c(cc1NC(=O)CN1C(=O)NC3(CCc4ccccc43)C1=O)oc1ccccc12"),
...     ("O=C1/C(=C/NC2CCS(=O)(=O)C2)c2ccccc2C(=O)N1c1ccccc1",
...      "O=C1/C(=C/NC2CCS(=O)(=O)C2)c2ccccc2C(=O)N1c1ccccc1"),
...     ("NC(=O)NC(Cc1ccccc1)C(=O)O",
...     "NC(=O)NC(Cc1ccccc1)C(=O)O")]
>>> train_smiles = [s[0] for s in data]
>>> tokens = set()
>>> for s in train_smiles:
...     tokens = tokens.union(set(c for c in s))
>>> tokens = sorted(list(tokens))
>>> from deepchem.models.optimizers import Adam, ExponentialDecay
>>> max_length = max(len(s) for s in train_smiles)
>>> batch_size = 100
>>> batches_per_epoch = len(train_smiles) / batch_size
>>> model = SeqToSeqModel(
...     tokens,
...     tokens,
...     max_length,
...     encoder_layers=2,
...     decoder_layers=2,
...     embedding_dimension=256,
...     model_dir="fingerprint",
...     batch_size=batch_size,
...     learning_rate=ExponentialDecay(0.001, 0.9, batches_per_epoch))
>>> for i in range(20):
...     loss = model.fit_sequences(data)
>>> prediction = model.predict_from_sequences(train_smiles, 5)
```

References

__init__(*input_tokens: List*, *output_tokens: List*, *max_output_length: int*, *encoder_layers: int = 4*, *decoder_layers: int = 4*, *batch_size: int = 100*, *embedding_dimension: int = 512*, *dropout: float = 0.0*, *reverse_input: bool = True*, *variational: bool = False*, *annealing_start_step: int = 5000*, *annealing_final_step: int = 10000*, ***kwargs*)

Construct a SeqToSeq model.

Parameters

- **input_tokens** (`list`) – List of all tokens that may appear in input sequences.
- **output_tokens** (`list`) – List of all tokens that may appear in output sequences
- **max_output_length** (`int`) – Maximum length of output sequence that may be generated
- **encoder_layers** (`int (default 4)`) – Number of recurrent layers in the encoder

- **decoder_layers** (*int (default 4)*) – Number of recurrent layers in the decoder
- **embedding_dimension** (*int (default 512)*) – Width of the embedding vector. This also is the width of all recurrent layers.
- **dropout** (*float (default 0.0)*) – Dropout probability to use during training.
- **reverse_input** (*bool (default True)*) – If True, reverse the order of input sequences before sending them into the encoder. This can improve performance when working with long sequences.
- **variational** (*bool (default False)*) – If True, train the model as a variational autoencoder. This adds random noise to the encoder, and also constrains the embedding to follow a unit Gaussian distribution.
- **annealing_start_step** (*int (default 5000)*) – Step (that is, batch) at which to begin turning on the constraint term for KL cost annealing.
- **annealing_final_step** (*int (default 10000)*) – Step (that is, batch) at which to finish turning on the constraint term for KL cost annealing.

fit_sequences(*sequences: List[str]*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *restore: bool = False*)

Train this model on a set of sequences

Parameters

- **sequences** (*List[str]*) – Training samples to fit to. Each sample should be represented as a tuple of the form (input_sequence, output_sequence).
- **max_checkpoints_to_keep** (*int*) – Maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – Frequency at which to write checkpoints, measured in training steps.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

predict_from_sequences(*sequences: List[str]*, *beam_width=5*)

Given a set of input sequences, predict the output sequences.

The prediction is done using a beam search with length normalization.

Parameters

- **sequences** (*List[str]*) – Input sequences to generate a prediction for
- **beam_width** (*int (default 5)*) – Beam width to use for searching. Set to 1 to use a simple greedy search.

predict_embedding(*sequences: List[str]*)

Given a set of input sequences, compute the embedding vectors.

Parameters

sequences (*List[str]*) – Input sequences to generate embeddings for.

predict_from_embedding(*embeddings: List[ndarray]*, *beam_width=5*)

Given a set of embedding vectors, predict the output sequences.

The prediction is done using a beam search with length normalization.

Parameters

- **embeddings** (*List[np.ndarray]*) – Embedding vectors to generate predictions for
- **beam_width** (*int*) – Beam width to use for searching. Set to 1 to use a simple greedy search.

3.18.26 GAN

class GAN(*noise_input_shape: tuple*, *data_input_shape: list*, *conditional_input_shape: list*, *generator_fn: Callable*, *discriminator_fn: Callable*, *device: device*, *n_generators: int = 1*, *n_discriminators: int = 1*, *create_discriminator_loss: Callable | None = None*, *create_generator_loss: Callable | None = None*, *_call_discriminator: Callable | None = None*, ***kwargs*)

Builder class for Generative Adversarial Networks.

A Generative Adversarial Network (GAN) [gan1] is a type of generative model. It consists of two parts called the "generator" and the "discriminator". The generator takes random noise as input and transforms it into an output that (hopefully) resembles the training data. The discriminator takes a set of samples as input and tries to distinguish the real training samples from the ones created by the generator. Both of them are trained together. The discriminator tries to get better and better at telling real from false data, while the generator tries to get better and better at fooling the discriminator.

Examples

Importing necessary modules

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.gan import GAN
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
```

Creating a Generator

```
>>> class Generator(nn.Module):
...     def __init__(self, noise_input_shape, conditional_input_shape):
...         super(Generator, self).__init__()
...         self.noise_input_shape = noise_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         self.noise_dim = noise_input_shape[1:]
...         self.conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(self.noise_dim) + sum(self.conditional_dim)
...         self.output = nn.Linear(input_dim, 1)
...     def forward(self, input):
...         noise_input, conditional_input = input
...         inputs = torch.cat((noise_input, conditional_input), dim=1)
...         output = self.output(inputs)
...         return output
```

Creating a Discriminator

```
>>> class Discriminator(nn.Module):
...     def __init__(self, data_input_shape, conditional_input_shape):
...         super(Discriminator, self).__init__()
...         self.data_input_shape = data_input_shape
```

(continues on next page)

(continued from previous page)

```
...         self.conditional_input_shape = conditional_input_shape
...         # Extracting the actual data dimension
...         data_dim = data_input_shape[1:]
...         # Extracting the actual conditional dimension
...         conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(data_dim) + sum(conditional_dim)
...         # Define the dense layers
...         self.dense1 = nn.Linear(input_dim, 10)
...         self.dense2 = nn.Linear(10, 1)
...     def forward(self, input):
...         data_input, conditional_input = input
...         # Concatenate data_input and conditional_input along the second␣
↪dimension
...         discrim_in = torch.cat((data_input, conditional_input), dim=1)
...         # Pass the concatenated input through the dense layers
...         x = F.relu(self.dense1(discrim_in))
...         output = torch.sigmoid(self.dense2(x))
...         return output
```

Defining an Example GAN class

```
>>> class ExampleGAN(dc.models.torch_models.GAN):
...    def get_noise_input_shape(self):
...        return (16,2,)
...    def get_data_input_shapes(self):
...        return [(16,1,)]
...    def get_conditional_input_shapes(self):
...        return [(16,1,)]
...    def create_generator(self):
...        noise_dim = self.get_noise_input_shape()
...        conditional_dim = self.get_conditional_input_shapes()[0]
...        return nn.Sequential(Generator(noise_dim, conditional_dim))
...    def create_discriminator(self):
...        data_input_shape = self.get_data_input_shapes()[0]
...        conditional_input_shape = self.get_conditional_input_shapes()[0]
...        return nn.Sequential(
...            Discriminator(data_input_shape, conditional_input_shape))
```

Defining the GAN

```
>>> batch_size = 16
>>> noise_shape = (batch_size, 2,)
>>> data_shape = [(batch_size, 1,)]
>>> conditional_shape = [(batch_size, 1,)]
>>> def create_generator(noise_dim, conditional_dim):
...     noise_dim = noise_dim
...     conditional_dim = conditional_dim[0]
...     return nn.Sequential(Generator(noise_dim, conditional_dim))
>>> def create_discriminator(data_input_shape, conditional_input_shape):
...     data_input_shape = data_input_shape[0]
...     conditional_input_shape = conditional_input_shape[0]
...     return nn.Sequential(
```

(continues on next page)

(continued from previous page)

```
...         Discriminator(data_input_shape, conditional_input_shape))
>>> gan = ExampleGAN(noise_shape,
...              data_shape,
...              conditional_shape,
...              create_generator(noise_shape, conditional_shape),
...              create_discriminator(data_shape, conditional_shape),
...              device='cpu')
>>> noise = torch.rand(*gan.noise_input_shape)
>>> real_data = torch.rand(*gan.data_input_shape[0])
>>> conditional = torch.rand(*gan.conditional_input_shape[0])
>>> gen_loss, disc_loss = gan([noise, real_data, conditional])
```

References

__init__(*noise_input_shape: tuple*, *data_input_shape: list*, *conditional_input_shape: list*, *generator_fn: Callable*, *discriminator_fn: Callable*, *device: device*, *n_generators: int = 1*, *n_discriminators: int = 1*, *create_discriminator_loss: Callable | None = None*, *create_generator_loss: Callable | None = None*, *_call_discriminator: Callable | None = None*, ***kwargs*)

Construct a GAN.

In addition to the parameters listed below, this class accepts all the keyword arguments from KerasModel.

Parameters

- **noise_input_shape** (*tuple*) – the shape of the noise input to the generator. The first dimension (corresponding to the batch size) should be omitted.
- **data_input_shape** (*list of tuple*) – the shapes of the inputs to the discriminator. The first dimension (corresponding to the batch size) should be omitted.
- **conditional_input_shape** (*list of tuple*) – the shapes of the conditional inputs to the generator and discriminator. The first dimension (corresponding to the batch size) should be omitted. If there are no conditional inputs, this should be an empty list.
- **generator_fn** (*Callable*) – a function that returns a generator. It will be called with no arguments. The returned value should be a nn.Module whose input is a list containing a batch of noise, followed by any conditional inputs. The number and shapes of its outputs must match the return value from get_data_input_shapes(), since generated data must have the same form as training data.
- **discriminator_fn** (*Callable*) – a function that returns a discriminator. It will be called with no arguments. The returned value should be a nn.Module whose input is a list containing a batch of data, followed by any conditional inputs. Its output should be a one dimensional tensor containing the probability of each sample being a training sample.
- **device** (*torch.device*) – the device to use for training
- **n_generators** (*int*) – the number of generators to include
- **n_discriminators** (*int*) – the number of discriminators to include
- **create_discriminator_loss** (*Callable*) – a function that returns the loss function for the discriminator. It will be called with two arguments: the output from the discriminator on a batch of training data, and the output from the discriminator on a batch of generated data. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

- **create_generator_loss** (*Callable*) – a function that returns the loss function for the generator. It will be called with one argument: the output from the discriminator on a batch of generated data. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.
- **_call_discriminator** (*Callable*) – a function that invokes the discriminator on a set of inputs. It will be called with three arguments: the discriminator to invoke, the list of data inputs, and the list of conditional inputs. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

`forward`(*inputs*) → Tuple[Tensor, Tensor]

Compute the output of the GAN.

Parameters
inputs (*list of Tensor*) – the inputs to the GAN. The first element must be a batch of noise, followed by data inputs and any conditional inputs.

Returns

- **total_gen_loss** (*Tensor*) – the total loss for the generator
- **total_discrim_loss** (*Tensor*) – the total loss for the discriminator

`get_noise_batch`(*batch_size: int*) → ndarray

Get a batch of random noise to pass to the generator.

This should return a NumPy array whose shape matches the one returned by get_noise_input_shape(). The default implementation returns normally distributed values. Subclasses can override this to implement a different distribution.

Parameters
batch_size (*int*) – the number of samples to generate

Returns
random_noise – a batch of random noise

Return type
ndarray

`create_generator_loss`(*discrim_output: Tensor*) → Tensor

Create the loss function for the generator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters
discrim_output (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Returns
output – A Tensor equal to the loss function to use for optimizing the generator.

Return type
Tensor

`create_discriminator_loss`(*discrim_output_train: Tensor*, *discrim_output_gen: Tensor*) → Tensor

Create the loss function for the discriminator.

The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

Parameters

- **discrim_output_train** (*Tensor*) – the output from the discriminator on a batch of training data. This is its estimate of the probability that each sample is training data.
- **discrim_output_gen** (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Returns
output – A Tensor equal to the loss function to use for optimizing the discriminator.

Return type
Tensor

discrim_loss_fn(*outputs: List*, *labels: List[Tensor]*, *weights: List[Tensor]*) → Any

Function to get the discriminator loss from the fit_generator output

Parameters

- **outputs** (*list of Tensor*) – the output from the discriminator on a batch of training data. This is its estimate of the probability that each sample is training data.
- **labels** (*Tensor*) – the labels for the batch. These are ignored.
- **weights** (*Tensor*) – the weights for the batch. These are ignored.

Return type
the value of the discriminator loss from the fit_generator output.

gen_loss_fn(*outputs: List*, *labels: List[Tensor]*, *weights: List[Tensor]*) → Tensor

Function to get the Generator loss from the fit_generator output

Parameters

- **outputs** (*Tensor*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.
- **labels** (*Tensor*) – the labels for the batch. These are ignored.
- **weights** (*Tensor*) – the weights for the batch. These are ignored.

Return type
the value of the generator loss function for this input.

3.18.27 GANModel

class GANModel(*n_generators: int = 1*, *n_discriminators: int = 1*, *create_discriminator_loss: Callable | None = None*, *create_generator_loss: Callable | None = None*, *_call_discriminator: Callable | None = None*, *device: device | None = None*, ***kwargs*)

Implements Generative Adversarial Networks.

A Generative Adversarial Network (GAN) is a type of generative model. It consists of two parts called the "generator" and the "discriminator". The generator takes random noise as input and transforms it into an output that (hopefully) resembles the training data. The discriminator takes a set of samples as input and tries to distinguish the real training samples from the ones created by the generator. Both of them are trained together. The discriminator tries to get better and better at telling real from false data, while the generator tries to get better and better at fooling the discriminator.

In many cases there also are additional inputs to the generator and discriminator. In that case it is known as a Conditional GAN (CGAN), since it learns a distribution that is conditional on the values of those inputs. They are referred to as "conditional inputs".

Many variations on this idea have been proposed, and new varieties of GANs are constantly being proposed. This class tries to make it very easy to implement straightforward GANs of the most conventional types. At the same time, it tries to be flexible enough that it can be used to implement many (but certainly not all) variations on the concept.

To define a GAN, you must create a subclass that provides implementations of the following methods:

get_noise_input_shape() get_data_input_shapes() create_generator() create_discriminator()

If you want your GAN to have any conditional inputs you must also implement:

get_conditional_input_shapes()

The following methods have default implementations that are suitable for most conventional GANs. You can override them if you want to customize their behavior:

create_generator_loss() create_discriminator_loss() get_noise_batch()

This class allows a GAN to have multiple generators and discriminators, a model known as MIX+GAN. It is described in [2] This can lead to better models, and is especially useful for reducing mode collapse, since different generators can learn different parts of the distribution. To use this technique, simply specify the number of generators and discriminators when calling the constructor. You can then tell predict_gan_generator() which generator to use for predicting samples.

Examples

Importing necessary modules

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.gan import GAN
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
```

Creating a Generator

```
>>> class Generator(nn.Module):
...     def __init__(self, noise_input_shape, conditional_input_shape):
...         super(Generator, self).__init__()
...         self.noise_input_shape = noise_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         self.noise_dim = noise_input_shape[1:]
...         self.conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(self.noise_dim) + sum(self.conditional_dim)
...         self.output = nn.Linear(input_dim, 1)
...     def forward(self, input):
...         noise_input, conditional_input = input
...         inputs = torch.cat((noise_input, conditional_input), dim=1)
...         output = self.output(inputs)
...         return output
```

Creating a Discriminator

```
>>> class Discriminator(nn.Module):
...     def __init__(self, data_input_shape, conditional_input_shape):
...         super(Discriminator, self).__init__()
...         self.data_input_shape = data_input_shape
```

(continues on next page)

(continued from previous page)

```
...         self.conditional_input_shape = conditional_input_shape
...         # Extracting the actual data dimension
...         data_dim = data_input_shape[1:]
...         # Extracting the actual conditional dimension
...         conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(data_dim) + sum(conditional_dim)
...         # Define the dense layers
...         self.dense1 = nn.Linear(input_dim, 10)
...         self.dense2 = nn.Linear(10, 1)
...     def forward(self, input):
...         data_input, conditional_input = input
...         # Concatenate data_input and conditional_input along the second␣
↪dimension
...         discrim_in = torch.cat((data_input, conditional_input), dim=1)
...         # Pass the concatenated input through the dense layers
...         x = F.relu(self.dense1(discrim_in))
...         output = torch.sigmoid(self.dense2(x))
...         return output
```

Defining an Example GAN class

```
>>> class ExampleGANModel(dc.models.torch_models.GANModel):
...     def get_noise_input_shape(self):
...         return (100,2,)
...     def get_data_input_shapes(self):
...         return [(100,1,)]
...     def get_conditional_input_shapes(self):
...         return [(100,1,)]
...     def create_generator(self):
...         noise_dim = self.get_noise_input_shape()
...         conditional_dim = self.get_conditional_input_shapes()[0]
...         return nn.Sequential(Generator(noise_dim, conditional_dim))
...     def create_discriminator(self):
...         data_input_shape = self.get_data_input_shapes()[0]
...         conditional_input_shape = self.get_conditional_input_shapes()[0]
...         return nn.Sequential(
...             Discriminator(data_input_shape, conditional_input_shape))
```

Defining a function to generate data

```
>>> def generate_batch(batch_size):
...     means = 10 * np.random.random([batch_size, 1])
...     values = np.random.normal(means, scale=2.0)
...     return means, values
```

```
>>> def generate_data(gan, batches, batch_size):
...     for _ in range(batches):
...         means, values = generate_batch(batch_size)
...         batch = {
...             gan.data_inputs[0]: values,
...             gan.conditional_inputs[0]: means
...         }
```

(continues on next page)

(continued from previous page)

```
...         yield batch
```

Defining the GANModel

```
>>> batch_size = 100
>>> noise_shape = (batch_size, 2,)
>>> data_shape = [(batch_size, 1,)]
>>> conditional_shape = [(batch_size, 1,)]
>>> gan = ExampleGANModel(learning_rate=0.01)
>>> data = generate_data(gan, 500, 100)
>>> gan.fit_gan(data, generator_steps=0.5, checkpoint_interval=0)
>>> means = 10 * np.random.random([1000, 1])
>>> values = gan.predict_gan_generator(conditional_inputs=[means])
```

References

Notes

This class is a subclass of TorchModel. It accepts all the keyword arguments from TorchModel.

`__init__`(*n_generators: int = 1*, *n_discriminators: int = 1*, *create_discriminator_loss: Callable | None = None*, *create_generator_loss: Callable | None = None*, *_call_discriminator: Callable | None = None*, *device: device | None = None*, ***kwargs*)

Parameters

- **`n_generators`** (*`int`*) – the number of generators to include
- **`n_discriminators`** (*`int`*) – the number of discriminators to include
- **`create_discriminator_loss`** (*`Callable`*) – a function that returns the loss function for the discriminator. It will be called with two arguments: the output from the discriminator on a batch of training data, and the output from the discriminator on a batch of generated data. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.
- **`create_generator_loss`** (*`Callable`*) – a function that returns the loss function for the generator. It will be called with one argument: the output from the discriminator on a batch of generated data. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.
- **`_call_discriminator`** (*`Callable`*) – a function that invokes the discriminator on a set of inputs. It will be called with three arguments: the discriminator to invoke, the list of data inputs, and the list of conditional inputs. The default implementation is appropriate for most cases. Subclasses can override this if the need to customize it.

`get_noise_input_shape`()

Get the shape of the generator's noise input layer.

Subclasses must override this to return a tuple giving the shape of the noise input. The actual Input layer will be created automatically. The dimension corresponding to the batch size should be omitted.

`get_data_input_shapes`()

Get the shapes of the inputs for training data.

Subclasses must override this to return a list of tuples, each giving the shape of one of the inputs. The actual Input layers will be created automatically. This list of shapes must also match the shapes of the generator's outputs. The dimension corresponding to the batch size should be omitted.

`get_conditional_input_shapes()`

Get the shapes of any conditional inputs.

Subclasses may override this to return a list of tuples, each giving the shape of one of the conditional inputs. The actual Input layers will be created automatically. The dimension corresponding to the batch size should be omitted.

The default implementation returns an empty list, meaning there are no conditional inputs.

`create_generator()`

Create and return a generator.

Subclasses must override this to construct the generator. The returned value should be a tf.keras.Model whose inputs are a batch of noise, followed by any conditional inputs. The number and shapes of its outputs must match the return value from get_data_input_shapes(), since generated data must have the same form as training data.

`create_discriminator()`

Create and return a discriminator.

Subclasses must override this to construct the discriminator. The returned value should be a tf.keras.Model whose inputs are all data inputs, followed by any conditional inputs. Its output should be a one dimensional tensor containing the probability of each sample being a training sample.

`fit_gan`(*batches*, *generator_steps=1*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *restore=False*) → None

Train this model on data.

Parameters

- **batches** (*iterable*) – batches of data to train the discriminator on, each represented as a dict that maps Inputs to values. It should specify values for all members of data_inputs and conditional_inputs.
- **generator_steps** (*float*) – the number of training steps to perform for the generator for each batch. This can be used to adjust the ratio of training steps for the generator and discriminator. For example, 2.0 will perform two training steps for every batch, while 0.5 will only perform one training step for every two batches.
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in batches. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint before training it.

`predict_gan_generator`(*batch_size=1*, *noise_input=None*, *conditional_inputs=[]*, *generator_index=0*)

Use the GAN to generate a batch of samples.

Parameters

- **batch_size** (*int*) – the number of samples to generate. If either noise_input or conditional_inputs is specified, this argument is ignored since the batch size is then determined by the size of that argument.

- **noise_input** (*array*) – the value to use for the generator's noise input. If None (the default), get_noise_batch() is called to generate a random input, so each call will produce a new set of samples.
- **conditional_inputs** (*list of arrays*) – the values to use for all conditional inputs. This must be specified if the GAN has any conditional inputs.
- **generator_index** (*int*) – the index of the generator (between 0 and n_generators-1) to use for generating the samples.

Returns

- *An array (if the generator has only one output) or list of arrays (if it has*
- *multiple outputs) containing the generated samples.*

3.18.28 WGANModel

class WGANModel(*gradient_penalty: float = 10.0*, ***kwargs*)

Implements Wasserstein Generative Adversarial Networks.

This class implements Wasserstein Generative Adversarial Networks (WGANs) as described in Arjovsky et al., "Wasserstein GAN" [wgan1]. A WGAN is conceptually rather different from a conventional GAN, but in practical terms very similar. It reinterprets the discriminator (often called the "critic" in this context) as learning an approximation to the Earth Mover distance between the training and generated distributions. The generator is then trained to minimize that distance. In practice, this just means using slightly different loss functions for training the generator and discriminator.

WGANs have theoretical advantages over conventional GANs, and they often work better in practice. In addition, the discriminator's loss function can be directly interpreted as a measure of the quality of the model. That is an advantage over conventional GANs, where the loss does not directly convey information about the quality of the model.

The theory WGANs are based on requires the discriminator's gradient to be bounded. The original paper achieved this by clipping its weights. This class instead does it by adding a penalty term to the discriminator's loss, as described in [wgan2]. This is sometimes found to produce better results.

There are a few other practical differences between GANs and WGANs. In a conventional GAN, the discriminator's output must be between 0 and 1 so it can be interpreted as a probability. In a WGAN, it should produce an unbounded output that can be interpreted as a distance.

When training a WGAN, you also should usually use a smaller value for generator_steps. Conventional GANs rely on keeping the generator and discriminator "in balance" with each other. If the discriminator ever gets too good, it becomes impossible for the generator to fool it and training stalls. WGANs do not have this problem, and in fact the better the discriminator is, the easier it is for the generator to improve. It therefore usually works best to perform several training steps on the discriminator for each training step on the generator.

Examples

Importing necessary modules

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.gan import WGANModel
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
```

Creating a Generator

```
>>> class Generator(nn.Module):
...     def __init__(self, noise_input_shape, conditional_input_shape):
...         super(Generator, self).__init__()
...         self.noise_input_shape = noise_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         self.noise_dim = noise_input_shape[1:]
...         self.conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(self.noise_dim) + sum(self.conditional_dim)
...         self.output = nn.Linear(input_dim, 1)
...     def forward(self, input):
...         noise_input, conditional_input = input
...         inputs = torch.cat((noise_input, conditional_input), dim=1)
...         output = self.output(inputs)
...         return output
```

Creating a Discriminator

```
>>> class Discriminator(nn.Module):
...     def __init__(self, data_input_shape, conditional_input_shape):
...         super(Discriminator, self).__init__()
...         self.data_input_shape = data_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         # Extracting the actual data dimension
...         data_dim = data_input_shape[1:]
...         # Extracting the actual conditional dimension
...         conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(data_dim) + sum(conditional_dim)
...         # Define the dense layers
...         self.dense1 = nn.Linear(input_dim, 10)
...         self.dense2 = nn.Linear(10, 1)
...     def forward(self, input):
...         data_input, conditional_input = input
...         # Concatenate data_input and conditional_input along the second␣
↪dimension
...         discrim_in = torch.cat((data_input, conditional_input), dim=1)
...         # Pass the concatenated input through the dense layers
...         x = F.relu(self.dense1(discrim_in))
...         output = self.dense2(x)
...         return output
```

Creating an Example WGANModel class

```
>>> class ExampleWGAN(WGANModel):
...     def get_noise_input_shape(self):
...         return (100,2,)
...     def get_data_input_shapes(self):
...         return [(100,1,)]
...     def get_conditional_input_shapes(self):
...         return [(100,1,)]
...     def create_generator(self):
...         noise_dim = self.get_noise_input_shape()
...         conditional_dim = self.get_conditional_input_shapes()[0]
```

(continues on next page)

(continued from previous page)

```
...         return nn.Sequential(Generator(noise_dim, conditional_dim))
...     def create_discriminator(self):
...         data_input_shape = self.get_data_input_shapes()[0]
...         conditional_input_shape = self.get_conditional_input_shapes()[0]
...         return nn.Sequential(
...             Discriminator(data_input_shape, conditional_input_shape))
```

Defining a function to generate data

```
>>> def generate_batch(batch_size):
...     means = 10 * np.random.random([batch_size, 1])
...     values = np.random.normal(means, scale=2.0)
...     return means, values
>>> def generate_data(gan, batches, batch_size):
...     for _ in range(batches):
...         means, values = generate_batch(batch_size)
...         batch = {
...             gan.data_inputs[0]: values,
...             gan.conditional_inputs[0]: means
...         }
...         yield batch
```

Defining the WGANModel

```
>>> wgan = ExampleWGAN(learning_rate=0.01,
...              gradient_penalty=0.1)
>>> data = generate_data(wgan, 500, 100)
>>> wgan.fit_gan(data, generator_steps=0.1, checkpoint_interval=0)
>>> means = 10 * np.random.random([1000, 1])
>>> values = wgan.predict_gan_generator(conditional_inputs=[means])
```

References

`__init__`(*gradient_penalty: float = 10.0*, ***kwargs*)

Construct a WGAN.

In addition to the following, this class accepts all the keyword arguments from GAN and TorchModel.

Parameters

`gradient_penalty` (*`float default 10.0`*) – the magnitude of the gradient penalty loss

`create_generator_loss`(*discrim_output: Tensor*) → Tensor

Create the loss function for the generator.

Parameters

`discrim_output` (*`torch.Tensor`*) – the output from the discriminator on a batch of generated data. This is its estimate of the probability that each sample is training data.

Returns

A Tensor equal to the mean of the inputs

Return type

torch.Tensor

create_discriminator_loss(*discrim_output_train: List[Tensor]*, *discrim_output_gen: Tensor*) → Tensor

Create the loss function for the discriminator.

Parameters

- **discrim_output_train** (*List[Tensor]*) – the output from the discriminator on a batch of training data. This is its estimate of the probability that each sample is training data.
- **discrim_output_gen** (*Tensor*) – the output from the discriminator on a batch of generated data.

Returns

A Tensor equal to the loss function to use for optimizing the discriminator.

Return type

torch.Tensor

3.18.29 BasicMolGANModel

class BasicMolGANModel(*edges: int = 5*, *vertices: int = 9*, *nodes: int = 5*, *embedding_dim: int = 10*, *dropout_rate: float = 0.0*, *device: device | None = None*, ***kwargs*)

Model for de-novo generation of small molecules based on work of Nicola De Cao et al. [molgan1]. It uses a GAN directly on graph data and a reinforcement learning objective to induce the network to generate molecules with certain chemical properties. Utilizes WGAN infrastructure; uses adjacency matrix and node features as inputs. Inputs need to be one-hot representation.

Examples

Import necessary libraries and modules

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models import BasicMolGANModel as MolGAN
>>> from deepchem.models.optimizers import ExponentialDecay
>>> import torch
>>> import torch.nn.functional as F
```

Load dataset and featurize molecules We will use a small dataset for this example. We will be using *MolGan-Featurizer* to featurize the molecules.

```
>>> smiles = ['CCC', 'C1=CC=CC=C1', 'CNC' ]
>>> # create featurizer
>>> feat = dc.feat.MolGanFeaturizer()
>>> # featurize molecules
>>> features = feat.featurize(smiles)
>>> # Remove empty objects
>>> features = list(filter(lambda x: x is not None, features))
```

Create and train the model

```
>>> # create model
>>> gan = MolGAN(learning_rate=ExponentialDecay(0.001, 0.9, 5000))
>>> dataset = dc.data.NumpyDataset([x.adjacency_matrix for x in features],[x.node_
↪features for x in features])
>>> def iterbatches(epochs):
```

(continues on next page)

(continued from previous page)

```
...     for i in range(epochs):
...         for batch in dataset.iterbatches(batch_size=gan.batch_size, pad_
↪batches=True):
...             adjacency_tensor = F.one_hot(
...                     torch.Tensor(batch[0]).to(torch.int64),
...                     gan.edges).to(torch.float32)
...             node_tensor = F.one_hot(
...                     torch.Tensor(batch[1]).to(torch.int64),
...                     gan.nodes).to(torch.float32)
...             yield {gan.data_inputs[0]: adjacency_tensor, gan.data_
↪inputs[1]:node_tensor}
>>> # train model
>>> gan.fit_gan(iterbatches(8), generator_steps=0.2, checkpoint_interval=0)
```

You can change the above parameters to get better results. The above example is just a simple example to show how to use the model. You can try *iterbatches(1000)* for better results.

Now, let's generate some molecules using the trained model We will generate 10 molecules and then convert them to RDKit molecules.

```
>>> generated_data = gan.predict_gan_generator(10)
Generating 10 samples
>>> # convert graphs to RDKitmolecules
>>> nmols = feat.defeaturize(generated_data)
>>> print("{} molecules generated".format(len(nmols)))
10 molecules generated
```

You can increase the number of generated molecules by changing the parameter in *predict_gan_generator* function. Generated molecules are in the form of GraphMatrix. You can convert them to RDKit molecules using *defeaturize* function of MolGanFeaturizer.

Now, let's remove invalid molecules from the generated molecules.

```
>>> # remove invalid moles
>>> nmols = list(filter(lambda x: x is not None, nmols))
>>> print ("{} valid molecules".format(len(nmols)))
0 valid molecules
```

We can see that currently training is unstable and 0 is a common outcome. You can try training the model with different parameters to get better results.

References

__init__(*edges: int = 5*, *vertices: int = 9*, *nodes: int = 5*, *embedding_dim: int = 10*, *dropout_rate: float = 0.0*, *device: device | None = None*, ***kwargs*)

Initialize the model

Parameters

- **edges** (*int, default 5*) – Number of bond types includes BondType.Zero
- **vertices** (*int, default 9*) – Max number of atoms in adjacency and node features matrices
- **nodes** (*int, default 5*) – Number of atom types in node features matrix

- **embedding_dim** (*int, default 10*) – Size of noise input array
- **dropout_rate** (*float, default = 0.*) – Rate of dropout used across whole model
- **name** (*str, default ''*) – Name of the model

get_noise_input_shape() → Tuple[int, int]

Return shape of the noise input used in generator

Returns
Shape of the noise input

Return type
Tuple

get_data_input_shapes() → List

Return input shape of the discriminator

Returns
List of shapes used as an input for distriminator.

Return type
List

create_generator()

Create generator model. Take noise data as an input and processes it through number of dense and dropout layers. Then data is converted into two forms one used for training and other for generation of compounds. The model has two outputs:

1. edges
2. nodes

The format differs depending on intended use (training or sample generation). For sample generation use flag, sample_generation=True while calling generator i.e. gan.generators[0](noise_input, training=False, sample_generation=True). For training the model, set *sample_generation=False*

create_discriminator(*units: List[Tuple[int, int] | int] = [(128, 64), 64]*)

Create discriminator model based on MolGAN layers. Takes two inputs:

1. adjacency tensor, containing bond information
2. nodes tensor, containing atom information

The input vectors need to be in one-hot encoding format. Use MolGAN featurizer for that purpose. It will be simplified in the future release.

predict_gan_generator(*batch_size: int = 1*, *noise_input: List | Tensor | None = None*, *conditional_inputs: List = []*, *generator_index: int = 0*) → List[GraphMatrix]

Use the GAN to generate a batch of samples.

Parameters

- **batch_size** (*int*) – the number of samples to generate. If either noise_input or conditional_inputs is specified, this argument is ignored since the batch size is then determined by the size of that argument.
- **noise_input** (*array*) – the value to use for the generator's noise input. If None (the default), get_noise_batch() is called to generate a random input, so each call will produce a new set of samples.
- **conditional_inputs** (*list of arrays*) – NOT USED. the values to use for all conditional inputs. This must be specified if the GAN has any conditional inputs.

- **generator_index** (*int*) – NOT USED. the index of the generator (between 0 and n_generators-1) to use for generating the samples.

Returns

Returns a list of GraphMatrix object that can be converted into RDKit molecules using MolGANFeaturizer defeaturize function.

Return type

List[GraphMatrix]

3.18.30 Weave

class Weave(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn=<function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = 'relu', batch_normalize: bool = True, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100*)

A graph convolutional network(GCN) for either classification or regression. The network consists of the following sequence of layers:

- Weave feature modules
- Final convolution
- Weave Gather Layer
- A fully connected layer
- A Softmax layer

Example

```
>>> import numpy as np
>>> import deepchem as dc
>>> featurizer = dc.feat.WeaveFeaturizer()
>>> X = featurizer(["C", "CC"])
>>> y = np.array([1, 0])
>>> batch_size = 2
>>> weavemodel = dc.models.WeaveModel(n_tasks=1,n_weave=2, fully_connected_layer_
↪sizes=[2000, 1000],mode="classification",batch_size=batch_size)
>>> atom_feat, pair_feat, pair_split, atom_split, atom_to_pair = weavemodel.compute_
↪features_on_batch(X)
>>> model = Weave(n_tasks=1,n_weave=2,fully_connected_layer_sizes=[2000, 1000],mode=
↪"classification")
>>> input_data = [atom_feat, pair_feat, pair_split, atom_split, atom_to_pair]
>>> output = model(input_data)
```

References

`__init__`(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn=<function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = 'relu', batch_normalize: bool = True, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_atom_feat** (*int, optional (default 75)*) – Number of features per atom. Note this is 75 by default and should be 78 if chirality is used by *WeaveFeaturizer*.
- **n_pair_feat** (*int, optional (default 14)*) – Number of features per pair of atoms.
- **n_hidden** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_graph_feat** (*int, optional (default 128)*) – Number of output features for each molecule(graph)
- **n_weave** (*int, optional (default 2)*) – The number of weave layers in this model.
- **fully_connected_layer_sizes** (list (default *[2000, 100]*)) – The size of each dense layer in the network. The length of this list determines the number of layers.
- **conv_weight_init_stddevs** (*list or float (default 0.03)*) – The standard deviation of the distribution to use for weight initialization of each convolutional layer. The length of this lisst should equal *n_weave*. Alternatively, this may be a single value instead of a list, in which case the same value is used for each layer.
- **weight_init_stddevs** (*list or float (default 0.01)*) – The standard deviation of the distribution to use for weight initialization of each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **bias_init_consts** (*list or float (default 0.0)*) – The value to initialize the biases in each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **dropouts** (*list or float (default 0.25)*) – The dropout probablity to use for each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **final_conv_activation_fn** (Optional[ActivationFn] (default *F.tanh*)) – The activation funcntion to apply to the final convolution at the end of the weave convolutions. If *None*, then no activate is applied (hence linear).
- **activation_fns** (str (default *relu*)) – The activation function to apply to each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

- **batch_normalize** (*bool, optional (default True)*) – If this is turned on, apply batch normalization before applying activation functions on convolutional and fully connected layers.
- **gaussian_expand** (*boolean, optional (default True)*) – Whether to expand each dimension of atomic features by gaussian histogram
- **compress_post_gaussian_expansion** (*bool, optional (default False)*) – If True, compress the results of the Gaussian expansion back to the original dimensions of the input.
- **mode** (*str (default "classification")*) – Either "classification" or "regression" for type of model.
- **n_classes** (*int (default 2)*) – Number of classes to predict (only used in classification mode)
- **batch_size** (*int (default 100)*) – Batch size used by this model for training.

forward(*inputs: Tensor | Sequence[Tensor]*) → List[Tensor]

Parameters
inputs (*OneOrMany[torch.Tensor]*) – Should contain 5 tensors [atom_features, pair_features, pair_split, atom_split, atom_to_pair]

Returns
Output as per use case : regression/classification

Return type
List[torch.Tensor]

3.18.31 WeaveModel

class WeaveModel(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn: ~typing.Callable | str | None = <function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = 'relu', batch_normalize: bool = True, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100, **kwargs*)

Implements Google-style Weave Graph Convolutions

This model implements the Weave style graph convolutions from [1]_.

The biggest difference between WeaveModel style convolutions and GraphConvModel style convolutions is that Weave convolutions model bond features explicitly. This has the side effect that it needs to construct a NxN matrix explicitly to model bond interactions. This may cause scaling issues, but may possibly allow for better modeling of subtle bond effects.

Note that [1]_ introduces a whole variety of different architectures for Weave models. The default settings in this class correspond to the W2N2 variant from [1]_ which is the most commonly used variant..

Examples

Here's an example of how to fit a *WeaveModel* on a tiny sample dataset.

```
>>> import numpy as np
>>> import deepchem as dc
>>> featurizer = dc.feat.WeaveFeaturizer()
>>> X = featurizer(["C", "CC"])
>>> y = np.array([1, 0])
>>> dataset = dc.data.NumpyDataset(X, y)
>>> model = dc.models.WeaveModel(n_tasks=1, n_weave=2, fully_connected_layer_
↪sizes=[2000, 1000], mode="classification")
>>> loss = model.fit(dataset)
```

References

`__init__`(*n_tasks: int, n_atom_feat: int | ~typing.Sequence[int] = 75, n_pair_feat: int | ~typing.Sequence[int] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: ~typing.List[int] = [2000, 100], conv_weight_init_stddevs: float | ~typing.Sequence[float] = 0.03, weight_init_stddevs: float | ~typing.Sequence[float] = 0.01, bias_init_consts: float | ~typing.Sequence[float] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: float | ~typing.Sequence[float] = 0.25, final_conv_activation_fn: ~typing.Callable | str | None = <function tanh>, activation_fns: ~typing.Callable | str | ~typing.Sequence[~typing.Callable | str] = 'relu', batch_normalize: bool = True, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, mode: str = 'classification', n_classes: int = 2, batch_size: int = 100, **kwargs*)

Parameters

- **`n_tasks`** (*`int`*) – Number of tasks
- **`n_atom_feat`** (*`int, optional (default 75)`*) – Number of features per atom. Note this is 75 by default and should be 78 if chirality is used by *WeaveFeaturizer*.
- **`n_pair_feat`** (*`int, optional (default 14)`*) – Number of features per pair of atoms.
- **`n_hidden`** (*`int, optional (default 50)`*) – Number of units(convolution depths) in corresponding hidden layer
- **`n_graph_feat`** (*`int, optional (default 128)`*) – Number of output features for each molecule(graph)
- **`n_weave`** (*`int, optional (default 2)`*) – The number of weave layers in this model.
- **`fully_connected_layer_sizes`** (list (default *[2000, 100]*)) – The size of each dense layer in the network. The length of this list determines the number of layers.
- **`conv_weight_init_stddevs`** (*`list or float (default 0.03)`*) – The standard deviation of the distribution to use for weight initialization of each convolutional layer. The length of this lisst should equal *n_weave*. Alternatively, this may be a single value instead of a list, in which case the same value is used for each layer.
- **`weight_init_stddevs`** (*`list or float (default 0.01)`*) – The standard deviation of the distribution to use for weight initialization of each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.

- **bias_init_consts** (*list or float (default 0.0)*) – The value to initialize the biases in each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **weight_decay_penalty** (*float (default 0.0)*) – The magnitude of the weight decay penalty to use
- **weight_decay_penalty_type** (*str (default "l2")*) – The type of penalty to use for weight decay, either 'l1' or 'l2'
- **dropouts** (*list or float (default 0.25)*) – The dropout probablity to use for each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **final_conv_activation_fn** (Optional[ActivationFn] (default *F.tanh*)) – The activation funcntion to apply to the final convolution at the end of the weave convolutions. If *None*, then no activate is applied (hence linear).
- **activation_fns** (str (default *relu*)) – The activation function to apply to each fully connected layer. The length of this list should equal len(layer_sizes). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **batch_normalize** (*bool, optional (default True)*) – If this is turned on, apply batch normalization before applying activation functions on convolutional and fully connected layers.
- **gaussian_expand** (*boolean, optional (default True)*) – Whether to expand each dimension of atomic features by gaussian histogram
- **compress_post_gaussian_expansion** (*bool, optional (default False)*) – If True, compress the results of the Gaussian expansion back to the original dimensions of the input.
- **mode** (*str (default "classification")*) – Either "classification" or "regression" for type of model.
- **n_classes** (*int (default 2)*) – Number of classes to predict (only used in classification mode)
- **batch_size** (*int (default 100)*) – Batch size used by this model for training.

compute_features_on_batch(*X_b*)

Compute tensors that will be input into the model from featurized representation.

The featurized input to *WeaveModel* is instances of *WeaveMol* created by *WeaveFeaturizer*. This method converts input *WeaveMol* objects into tensors used by the Keras implementation to compute *WeaveModel* outputs.

Parameters

X_b (*np.ndarray*) – A numpy array with dtype=object where elements are *WeaveMol* objects.

Returns

- **atom_feat** (*np.ndarray*) – Of shape *(N_atoms, N_atom_feat)*.
- **pair_feat** (*np.ndarray*) – Of shape *(N_pairs, N_pair_feat)*. Note that *N_pairs* will depend on the number of pairs being considered. If *max_pair_distance* is *None*, then this will be *N_atoms**2*. Else it will be the number of pairs within the specifed graph distance.

- **pair_split** (*np.ndarray*) – Of shape *(N_pairs,)*. The i-th entry in this array will tell you the originating atom for this pair (the "source"). Note that pairs are symmetric so for a pair *(a, b)*, both *a* and *b* will separately be sources at different points in this array.
- **atom_split** (*np.ndarray*) – Of shape *(N_atoms,)*. The i-th entry in this array will be the molecule with the i-th atom belongs to.
- **atom_to_pair** (*np.ndarray*) – Of shape *(N_pairs, 2)*. The i-th row in this array will be the array *[a, b]* if *(a, b)* is a pair to be considered. (Note by symmetry, this implies some other row will contain *[b, a]*.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Convert a dataset into the tensors needed for learning.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to convert
- **epochs** (`int, optional (Default 1)`) – Number of times to walk over *dataset*
- **mode** (`str, optional (Default 'fit')`) – Ignored in this implementation.
- **deterministic** (`bool, optional (Default True)`) – Whether the dataset should be walked in a deterministic fashion
- **pad_batches** (`bool, optional (Default True)`) – If true, each returned batch will have size *self.batch_size*.

Return type

Iterator which walks over the batches

3.18.32 ProgressiveMultitaskModel

class ProgressiveMultitaskModel(*n_tasks: int*, *n_features: int*, *layer_sizes: List[int] = [1000]*, *mode: Literal['regression', 'classification'] = 'regression'*, *alpha_init_stddevs: float | Sequence[float] = 0.02*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *dropouts: float | Sequence[float] = 0.5*, *n_classes: int | None = None*, *n_outputs: int | None = None*, ***kwargs*)

Implements a progressive multitask neural network in PyTorch.

Progressive networks allow for multitask learning where each task gets a new column of weights and lateral connections to previous tasks are added to the network. As a result, there is no exponential forgetting where previous tasks are ignored.

Examples

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models import ProgressiveMultitaskModel
>>> featurizer = dc.feat.CircularFingerprint(size=1024, radius=4)
>>> tasks, datasets, transformers = dc.molnet.load_tox21(featurizer=featurizer)
>>> train_dataset, valid_dataset, test_dataset = datasets
>>> n_tasks = len(tasks)
>>> model = ProgressiveMultitaskModel(n_tasks, 1024, layer_sizes=[1024], mode=
↪'classification')
>>> model.fit(train_dataset, nb_epoch=10)
```

References

See [1]_ for a full description of the progressive architecture

`__init__`(*n_tasks: int*, *n_features: int*, *layer_sizes: List[int] = [1000]*, *mode: Literal['regression', 'classification'] = 'regression'*, *alpha_init_stddevs: float | Sequence[float] = 0.02*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *weight_decay_penalty: float = 0.0*, *weight_decay_penalty_type: str = 'l2'*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *dropouts: float | Sequence[float] = 0.5*, *n_classes: int | None = None*, *n_outputs: int | None = None*, ***kwargs*)

Parameters

- **n_tasks** (*int*) – Number of tasks.
- **n_features** (*int*) – Size of input feature vector.
- **layer_sizes** (*list of ints*) – List of layer sizes.
- **mode** (*str*) – Type of model. Must be 'regression' or 'classification'.
- **alpha_init_stddevs** (*float or list of floats*) – Standard deviation for truncated normal distribution to initialize alpha parameters.
- **weight_init_stddevs** (*float or list of floats*) – Standard deviation for truncated normal distribution to initialize weight parameters.
- **bias_init_consts** (*float or list of floats*) – Constant value to initialize bias parameters.
- **weight_decay_penalty** (*float*) – Amount of weight decay penalty to use.
- **weight_decay_penalty_type** (*str*) – Type of weight decay penalty. Must be 'l1' or 'l2'.
- **activation_fns** (*str or list of str*) – Name of activation function(s) to use.
- **dropouts** (*float or list of floats*) – Dropout probability.
- **n_classes** (*int*) – The number of classes to predict per task. Default to 2 for classification and 1 for regression.
- **n_outputs** (*int*) – The number of outputs to predict per task. Deprecated, use n_classes instead.

`fit`(*dataset: Dataset*, *nb_epoch: int = 10*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *deterministic: bool = False*, *restore: bool = False*, *variables: List[Parameter] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*)

Train this model on a dataset.

Parameters

- **dataset** (Dataset) – the Dataset to train on
- **nb_epoch** (*int*) – the number of epochs to train for
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **deterministic** (*bool*) – if True, the samples are processed in order. If False, a different random order is used for each epoch.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step, **kwargs) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type

The average loss over the most recent checkpoint interval

fit_task(*dataset: Dataset*, *task: int*, *nb_epoch: int = 10*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *deterministic: bool = False*, *restore: bool = False*, *variables: List[Parameter] | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*)

Train this model on one task. Called by fit() to train each task sequentially. Calls fit_generator() internally.

Parameters

- **dataset** (Dataset) – the Dataset to train on
- **task** (*int*) – the task to train on
- **nb_epoch** (*int*) – the number of epochs to train for
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **deterministic** (*bool*) – if True, the samples are processed in order. If False, a different random order is used for each epoch.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type

The average loss over the most recent checkpoint interval

3.18.33 Density Functional Theory Model - XCModel

3.18.34 TextCNNModel

class TextCNNModel(*n_tasks: int*, *char_dict: Dict[str, int]*, *seq_length: int*, *n_embedding: int = 75*, *kernel_sizes: List[int] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20]*, *num_filters: List[int] = [100, 200, 200, 200, 200, 100, 100, 100, 100, 100, 160, 160]*, *dropout: float = 0.25*, *mode: str = 'classification'*, ***kwargs*)

A 1D convolutional neural network to work on smiles strings for both classification and regression tasks.

Reimplementation of the discriminator module in ORGAN [1] . Originated from [2].

The model converts the input smile strings to an embedding vector, the vector is convolved and pooled through a series of convolutional filters which are concatnated and later passed through a simple dense layer. The resulting vector goes through a Highway layer [3] which finally as per the nature of the task is passed through a dense layer.

References

Examples

```
>>> import os
>>> from deepchem.models.torch_models import TextCNNModel
>>> from deepchem.models.torch_models.text_cnn import default_dict
>>> n_tasks = 1
>>> seq_len = 250
>>> model = TextCNNModel(n_tasks, default_dict, seq_len)
```

__init__(*n_tasks: int*, *char_dict: Dict[str, int]*, *seq_length: int*, *n_embedding: int = 75*, *kernel_sizes: List[int] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20]*, *num_filters: List[int] = [100, 200, 200, 200, 200, 100, 100, 100, 100, 100, 160, 160]*, *dropout: float = 0.25*, *mode: str = 'classification'*, ***kwargs*) → None

Parameters

- **n_tasks** (*int*) – Number of tasks
- **char_dict** (*dict*) – Mapping from characters in smiles to integers

- **seq_length** (*int*) – Length of sequences(after padding)
- **n_embedding** (*int, optional*) – Length of embedding vector
- **filter_sizes** (*list of int, optional*) – Properties of filters used in the conv net
- **num_filters** (*list of int, optional*) – Properties of filters used in the conv net
- **dropout** (*float, optional*) – Dropout rate
- **mode** (*str*) – Either "classification" or "regression" for type of model.

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Transfer smiles strings to fixed length integer vectors

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to convert
- **epochs** (*int, optional (Default 1)*) – Number of times to walk over *dataset*
- **mode** (*str, optional (Default 'fit')*) – Ignored in this implementation.
- **deterministic** (*bool, optional (Default True)*) – Whether the dataset should be walked in a deterministic fashion
- **pad_batches** (*bool, optional (Default True)*) – If true, each returned batch will have size *self.batch_size*.

Return type

Iterator which walks over the batches

static build_char_dict(*dataset: Dataset*, *default_dict: Dict[str, int] = {'#': 1, '(': 2, ')': 3, '+': 4, '-': 5, '/': 6, '1': 7, '2': 8, '3': 9, '4': 10, '5': 11, '6': 12, '7': 13, '8': 14, '=': 15, 'Br': 30, 'C': 16, 'Cl': 29, 'F': 17, 'H': 18, 'I': 19, 'N': 20, 'O': 21, 'P': 22, 'S': 23, '[': 24, '\\': 25, ']': 26, '_': 27, 'c': 28, 'n': 31, 'o': 32, 's': 33}*)

Collect all unique characters(in smiles) from the dataset. This method should be called before defining the model to build appropriate char_dict

Parameters

- **dataset** (Dataset) – Dataset for which char_dict is built for
- **default_dict** (*dict, optional*) – Mapping from characters in smiles to integers, optional

Returns

- **out_dict** (*dict*) – A dictionary containing mapping between unique characters in the dataset to integers
- **seq_length** (*int*) – The maximum sequence length of smile strings found in the dataset multiplied by 1.2

smiles_to_seq(*smiles: str*)

Tokenize characters in smiles to integers

Parameters

smiles (*str*) – A smile string

Returns

array – An array of integers representing the tokenized sequence of characters.

Return type
np.ndarray

static convert_bytes_to_char(*s: bytes*) → str

Convert bytes to string.

Parameters
s (*bytes*) – Bytes to be converted to string.

Returns
String representation of the bytes.

Return type
str

smiles_to_seq_batch(*ids_b: List[bytes | str] | ndarray*) → ndarray

Converts SMILES strings to np.array sequence.

Parameters
ids_b (*Union[List[Union[bytes, str]], np.ndarray]*) – A list of SMILES strings, either as bytes or strings.

Returns
A numpy array containing the tokenized sequences of SMILES strings.

Return type
np.ndarray

3.18.35 UNetModel

class UNetModel(*in_channels: int = 3*, *out_channels: int = 1*, ***kwargs*)

UNet model for image segmentation.

UNet is a convolutional neural network architecture for fast and precise segmentation of images based on the works of Ronneberger et al. [1]. The architecture consists of an encoder, a bottleneck, and a decoder. The encoder downsamples the input image to capture the context of the image. The bottleneck captures the most important features of the image. The decoder upsamples the image to generate the segmentation mask. The encoder and decoder are connected by skip connections to preserve spatial information.

Examples

Importing necessary modules

```
>>> import numpy as np
>>> import deepchem as dc
>>> from deepchem.models.torch_models import UNet
```

Creating a random dataset of 5 32x32 pixel RGB input images and 5 32x32 pixel grey scale output images

```
>>> x = np.random.randn(5, 3, 32, 32).astype(np.float32)
>>> y = np.random.rand(5, 1, 32, 32).astype(np.float32)
>>> dataset = dc.data.NumpyDataset(x, y)
```

We will create a UNet model with 3 input channels and 1 output channel. We will then fit the model on the dataset for 5 epochs and predict the output images.

```
>>> model = UNetModel(in_channels=3, out_channels=1)
>>> model.fit(dataset, nb_epoch=5)
>>> predictions = model.predict(dataset)
```

Notes

1. This implementation of the UNet model makes some changes to the padding of the inputs to the convolutional layers. The padding is set to 'same' to ensure that the output size of the convolutional layers is the same as the input size. This is done to preserve the spatial information of the input image and to keep the output size of the encoder and decoder the same.

2. The input image size must be divisible by 2^4 = 16 to ensure that the output size of the encoder and decoder is the same.

References

`__init__`(*in_channels: int = 3*, *out_channels: int = 1*, ***kwargs*)

Parameters

- **`input_channels`** (*`int (default 3)`*) – Number of input channels.
- **`output_channels`** (*`int (default 1)`*) – Number of output channels.

3.19 PyTorch Lightning Models

DeepChem supports the use of PyTorch-Lightning to build PyTorch models.

3.19.1 DCLightningModule

You can wrap an arbitrary `TorchModel` in a `DCLightningModule` object.

class `DCLightningModule`(*dc_model*)

DeepChem Lightning Module to be used with Lightning trainer.

TODO: Add dataloader, example code and fit, once datasetmodule is ready The lightning module is a wrapper over deepchem's torch model. This module directly works with pytorch lightning trainer which runs training for multiple epochs and also is responsible for setting up and training models on multiple GPUs. https://pytorch-lightning.readthedocs.io/en/stable/api/pytorch_lightning.core.LightningModule.html?highlight=LightningModule

Notes

This class requires PyTorch to be installed.

`__init__`(*dc_model*)

Create a new DCLightningModule.

Parameters

`dc_model` (`deepchem.models.torch_models.torch_model.TorchModel`) – TorchModel to be wrapped inside the lightning module.

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple. Optimization with multiple optimizers only works in the manual optimization mode.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
```

(continues on next page)

(continued from previous page)

```
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated",
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` automatically in case of automatic optimization.
- If a learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizer.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, you will have to switch to 'manual optimization' mode and step them yourself.
- If you need to control how often the optimizer steps, override the `optimizer_step()` hook.

`training_step`(*batch*, *batch_idx*)

Perform a training step.

Parameters

- **batch** (*A tensor, tuple or list.*) –
- **batch_idx** (*Integer displaying index of this batch*) –

- **optimizer_idx** (*When using multiple optimizers, this argument will also be present.*) –

Returns
loss_outputs

Return type
outputs of losses.

3.20 Jax Models

DeepChem supports the use of Jax to build deep learning models.

3.20.1 JaxModel

class JaxModel(*forward_fn: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], params: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], loss: ~deepchem.models.losses.Loss | ~typing.Callable[[~typing.List, ~typing.List, ~typing.List], ~typing.Any] | None, output_types: ~typing.List[str] | None = None, batch_size: int = 100, learning_rate: float = 0.001, optimizer: ~optax._src.base.GradientTransformation | ~deepchem.models.optimizers.Optimizer | None = None, grad_fn: ~typing.Callable = <function create_default_gradient_fn>, update_fn: ~typing.Callable = <function create_default_update_fn>, eval_fn: ~typing.Callable = <function create_default_eval_fn>, rng=Array([0, 1], dtype=uint32), log_frequency: int = 100, **kwargs*)

This is a DeepChem model implemented by a Jax Model Here is a simple example of that uses JaxModel to train a Haiku (JAX Neural Network Library) based model on deepchem dataset.

```
>>>
>> def forward_model(x):
>>   net = hk.nets.MLP([512, 256, 128, 1])
>>   return net(x)
>> def rms_loss(pred, tar, w):
>>   return jnp.mean(optax.l2_loss(pred, tar))
>> params_init, forward_fn = hk.transform(forward_model)
>> rng = jax.random.PRNGKey(500)
>> inputs, _, _, _ = next(iter(dataset.iterbatches(batch_size=256)))
>> params = params_init(rng, inputs)
>> j_m = JaxModel(forward_fn, params, rms_loss, 256, 0.001, 100)
>> j_m.fit(train_dataset)
```

All optimizations will be done using the optax library.

__init__(*forward_fn: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], params: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], loss: ~deepchem.models.losses.Loss | ~typing.Callable[[~typing.List, ~typing.List, ~typing.List], ~typing.Any] | None, output_types: ~typing.List[str] | None = None, batch_size: int = 100, learning_rate: float = 0.001, optimizer: ~optax._src.base.GradientTransformation | ~deepchem.models.optimizers.Optimizer | None = None, grad_fn: ~typing.Callable = <function create_default_gradient_fn>, update_fn: ~typing.Callable = <function create_default_update_fn>, eval_fn: ~typing.Callable = <function create_default_eval_fn>, rng=Array([0, 1], dtype=uint32), log_frequency: int = 100, **kwargs*)

Create a new JaxModel

Parameters

- **model** (*hk.State or Function*) – Any Jax based model that has a *apply* method for computing the network. Currently only haiku models are supported.
- **params** (*hk.Params*) – The parameter of the Jax based networks
- **loss** (*dc.models.losses.Loss or function*) – a Loss or function defining how to compute the training loss for each batch, as described above
- **output_types** (*list of strings, optional (default None)*) – the type of each output from the model, as described above
- **batch_size** (*int, optional (default 100)*) – default batch size for training and evaluating
- **learning_rate** (*float or* LearningRateSchedule*, optional (default 0.001)*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.
- **optimizer** (*optax object*) – For the time being, it is optax object
- **rng** (*jax.random.PRNGKey, optional (default 1)*) – A default global PRNG key to use for drawing random numbers.
- **log_frequency** (*int, optional (default 100)*) – The frequency at which to log data. Data is logged using *logging* by default.

Miscellanous Parameters Yet To Add

model_dir: str, optional (default None)
Will be added along with the save & load method

tensorboard: bool, optional (default False)
whether to log progress to TensorBoard during training

wandb: bool, optional (default False)
whether to log progress to Weights & Biases during training

Work in Progress

[1] Integrate the optax losses, optimizers, schedulers with Deepchem [2] Support for saving & loading the model.

fit(*dataset:* Dataset, *nb_epochs: int = 10*, *deterministic: bool = False*, *loss:* Loss *| Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on a dataset. :param dataset: the Dataset to train on :type dataset: Dataset :param nb_epoch: the number of epochs to train for :type nb_epoch: int :param deterministic: if True, the samples are processed in order. If False, a different random

order is used for each epoch.

Parameters

- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step) that will be invoked after every step. This can be used to perform validation, logging, etc.

- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Returns

- *The average loss over the most recent checkpoint interval*
- *Miscellanous Parameters Yet To Add*
- ———————————-
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of hk.Variable*) – the variables to train. If None (the default), all trainable variables in the model are used.
- *Work in Progress*
- —————-
- *[1] Integerate the optax losses, optimizers, schedulers with Deepchem*
- *[2] Support for saving & loading the model.*
- *[3] Adding support for output types (choosing only self._loss_outputs)*

predict_on_generator(*generator: Iterable[Tuple[Any, Any, Any]], transformers: List[*Transformer*] = []*, *output_types: str | Sequence[str] | None = None*) → ndarray | Sequence[ndarray]

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **transformers** (*List[dc.trans.Transformers]*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

predict_on_batch(*X: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *transformers: List[*Transformer*] = []*) → ndarray | Sequence[ndarray]

Generates predictions for input samples, processing samples in a batch. :param X: the input data, as a Numpy array. :type X: ndarray :param transformers: Transformers that the input data has been transformed by. The output

is passed through these transformers to undo the transformations.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

predict(*dataset: Dataset*, *transformers: List[Transformer] = []*, *output_types: List[str] | None = None*) → ndarray | Sequence[ndarray]

Uses self to make predictions on provided Dataset object.

Parameters

- **dataset** (*dc.data.Dataset*) – Dataset to make prediction on
- **transformers** (*List[dc.trans.Transformers]*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **output_types** (*String or list of Strings*) – If specified, all outputs of this type will be retrieved from the model. If output_types is specified, outputs must be None.

Returns

- *a NumPy array of the model produces a single output, or a list of arrays*
- *if it produces multiple outputs*

get_global_step() → int

Get the number of steps of fitting that have been performed.

evaluate_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *metrics: List[Metric]*, *transformers: List[Transformer] = []*, *per_task_metrics: bool = False*)

Evaluate the performance of this model on the data produced by a generator. :param generator: this should generate batches, each represented as a tuple of the form

(inputs, labels, weights).

Parameters

- **metric** (*list of* deepchem.metrics.Metric) – Evaluation metric
- **transformers** (*List[dc.trans.Transformers]*) – Transformers that the input data has been transformed by. The output is passed through these transformers to undo the transformations.
- **per_task_metrics** (*bool*) – If True, return per-task scores.

Returns
Maps tasks to scores under metric.

Return type
dict

default_generator(*dataset: Dataset*, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset. Subclasses may override this method to customize how model inputs are generated from the data. :param dataset: the data to iterate :type dataset: Dataset :param epochs: the number of times to iterate over the full dataset :type epochs: int :param mode: allowed values are 'fit' (called during training), 'predict' (called

during prediction), and 'uncertainty' (called during uncertainty prediction)

Parameters

- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.20.2 PinnModel

class PINNModel(*forward_fn: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], params: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], initial_data: dict = {}, output_types: ~typing.List[str] | None = None, batch_size: int = 100, learning_rate: float = 0.001, optimizer: ~optax._src.base.GradientTransformation | ~deepchem.models.optimizers.Optimizer | None = None, grad_fn: ~typing.Callable = <function create_default_gradient_fn>, update_fn: ~typing.Callable = <function create_default_update_fn>, eval_fn: ~typing.Callable = <function create_default_eval_fn>, rng=Array([0, 1], dtype=uint32), log_frequency: int = 100, **kwargs*)

This is class is derived from the JaxModel class and methods are also very similar to JaxModel, but it has the option of passing multiple arguments(Done using *args) suitable for PINNs model. Ex - Approximating f(x, y, z, t) satisfying a Linear differential equation.

This model is recommended for linear partial differential equations but if you can accurately write the gradient function in Jax depending on your use case, then it will work as well.

This class requires two functions apart from the usual function definition and weights

[1] **grad_fn** : Each PINNs have a different strategy for calculating its final losses. This function tells the PINNModel how to go about computing the derivatives for backpropagation. It should follow this format:

```
>>>
>> def gradient_fn(forward_fn, loss_outputs, initial_data):
>>
>>  def model_loss(params, target, weights, rng, ...):
>>
>>    # write code using the arguments.
>>    # ... indicates the variable number of positional arguments.
>>    return
>>
>>  return model_loss
```

"..." can be replaced with various arguments like (x, y, z, y) but should match with eval_fn

[2] **eval_fn**: Function for defining how the model needs to compute during inference. It should follow this format

```
>>>
>> def create_eval_fn(forward_fn, params):
>>  def eval_model(..., rng=None):
>>    # write code here using arguments
>>
>>    return
>>  return eval_model
```

"…" can be replaced with various arguments like (x, y, z, y) but should match with grad_fn

[3] boundary_data: For a detailed example, check out - deepchem/models/jax_models/tests/test_pinn.py where we have solved f'(x) = -sin(x)

References

Notes

This class requires Jax, Haiku and Optax to be installed.

`__init__`(*forward_fn: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], params: ~collections.abc.Mapping[str, ~collections.abc.Mapping[str, ~jax.Array]], initial_data: dict = {}, output_types: ~typing.List[str] | None = None, batch_size: int = 100, learning_rate: float = 0.001, optimizer: ~optax._src.base.GradientTransformation | ~deepchem.models.optimizers.Optimizer | None = None, grad_fn: ~typing.Callable = <function create_default_gradient_fn>, update_fn: ~typing.Callable = <function create_default_update_fn>, eval_fn: ~typing.Callable = <function create_default_eval_fn>, rng=Array([0, 1], dtype=uint32), log_frequency: int = 100, **kwargs*)

Parameters

- **forward_fn** (*hk.State or Function*) – Any Jax based model that has a *apply* method for computing the network. Currently only haiku models are supported.
- **params** (*hk.Params*) – The parameter of the Jax based networks
- **initial_data** (*dict*) – This acts as a session variable which will be passed as a dictionary in grad_fn
- **output_types** (*list of strings, optional (default None)*) – the type of each output from the model, as described above
- **batch_size** (*int, optional (default 100)*) – default batch size for training and evaluating
- **learning_rate** (*float or* LearningRateSchedule*, optional (default 0.001)*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.
- **optimizer** (*optax object*) – For the time being, it is optax object
- **grad_fn** (*Callable (default create_default_gradient_fn)*) – It defines how the loss function and gradients need to be calculated for the PINNs model
- **update_fn** (*Callable (default create_default_update_fn)*) – It defines how the weights need to be updated using backpropogation. We have used optax library for optimisation operations. Its reccomended to leave this default.
- **eval_fn** (*Callable (default create_default_eval_fn)*) – Function for defining on how the model needs to compute during inference.
- **rng** (*jax.random.PRNGKey, optional (default 1)*) – A default global PRNG key to use for drawing random numbers.
- **log_frequency** (*int, optional (default 100)*) – The frequency at which to log data. Data is logged using *logging* by default.

`default_generator`(*dataset:* Dataset, *epochs: int = 1*, *mode: str = 'fit'*, *deterministic: bool = True*, *pad_batches: bool = True*) → Iterable[Tuple[List, List, List]]

Create a generator that iterates batches for a dataset. Subclasses may override this method to customize how model inputs are generated from the data.

Parameters

- **dataset** (Dataset) – the data to iterate
- **epochs** (*int*) – the number of times to iterate over the full dataset
- **mode** (*str*) – allowed values are 'fit' (called during training), 'predict' (called during prediction), and 'uncertainty' (called during uncertainty prediction)
- **deterministic** (*bool*) – whether to iterate over the dataset in order, or randomly shuffle the data for each epoch
- **pad_batches** (*bool*) – whether to pad each batch up to this model's preferred batch size

Returns

- *a generator that iterates batches, each represented as a tuple of lists*
- *([inputs], [outputs], [weights])*

3.21 Hugging Face Models

HuggingFace models from the transformers library can wrapped using the wrapper `HuggingFaceModel`

class HuggingFaceModel(*model: PreTrainedModel*, *tokenizer: transformers.tokenization_utils.PreTrainedTokenizer*, *task: str | None = None*, ***kwargs*)

Wrapper class that wraps HuggingFace models as DeepChem models

The class provides a wrapper for wrapping models from HuggingFace ecosystem in DeepChem and training it via DeepChem's api. The reason for this might be that you might want to do an apples-to-apples comparison between HuggingFace from the transformers library and DeepChem library.

The *HuggingFaceModel* has a Has-A relationship by wrapping models from *transformers* library. Once a model is wrapped, DeepChem's API are used for training, prediction, evaluation and other downstream tasks.

A *HuggingFaceModel* wrapper also has a *tokenizer* which tokenizes raw SMILES strings into tokens to be used by downstream models. The SMILES strings are generally stored in the *X* attribute of deepchem.data.Dataset object'. This differs from the DeepChem standard workflow as tokenization is done on the fly here. The approach allows us to leverage *transformers* library's fast tokenization algorithms and other utilities like data collation, random masking of tokens for masked language model training etc.

Parameters

- **model** (*transformers.modeling_utils.PreTrainedModel*) – The HuggingFace model to wrap.
- **task** (*str, (optional, default None)*) –

  The task defines the type of learning task in the model. The supported tasks are

  - *mlm* - masked language modeling commonly used in pretraining
  - *mtr* - multitask regression - a task used for both pretraining base models and finetuning
  - *regression* - use it for regression tasks, like property prediction
  - *classification* - use it for classification tasks

  When the task is not specified or None, the wrapper returns raw output of the HuggingFaceModel. In cases where the HuggingFaceModel is a model without a task specific head, this output will be the last hidden states.

- **tokenizer** (*transformers.tokenization_utils.PreTrainedTokenizer*) – Tokenizer

Example

```
>>> import os
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
```

```
>>> # preparing dataset
>>> smiles = ['CN(c1ccccc1)c1ccccc1C(=O)NCC1(O)CCOCC1',
↪'CC[NH+](CC)C1CCC([NH2+]C2CC2)(C(=O)[O-])C1', \
...     'COCC(CNC(=O)c1ccc2c(c1)NC(=O)C2)OC', 'OCCn1cc(CNc2cccc3c2CCCC3)nn1', \
...     'CCCCCCc1ccc(C#Cc2ccc(C#CC3=CC=C(CCC)CC3)c(C3CCCCC3)c2)c(F)c1',
↪'nO=C(NCc1ccc(F)cc1)N1CC=C(c2c[nH]c3ccccc23)CC1']
>>> filepath = os.path.join(tempdir, 'smiles.txt')
>>> f = open(filepath, 'w')
>>> f.write('\n'.join(smiles))
253
>>> f.close()
```

```
>>> # preparing tokenizer
>>> from tokenizers import ByteLevelBPETokenizer
>>> from transformers.models.roberta import RobertaTokenizerFast
>>> tokenizer = ByteLevelBPETokenizer()
>>> tokenizer.train(files=filepath, vocab_size=1_000, min_frequency=2, special_
↪tokens=["<s>", "<pad>", "</s>", "<unk>", "<mask>"])
>>> tokenizer_path = os.path.join(tempdir, 'tokenizer')
>>> os.makedirs(tokenizer_path)
>>> result = tokenizer.save_model(tokenizer_path)
>>> tokenizer = RobertaTokenizerFast.from_pretrained(tokenizer_path)
```

```
>>> # preparing dataset
>>> import pandas as pd
>>> import deepchem as dc
>>> smiles = ["CCN(CCSC)C(=O)N[C@@](C)(CC)C(F)(F)F",
↪"CC1(C)CN(C(=O)Nc2cc3ccccc3nn2)C[C@@]2(CCOC2)O1"]
>>> labels = [3.112,2.432]
>>> df = pd.DataFrame(list(zip(smiles, labels)), columns=["smiles", "task1"])
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_csv(tmpfile.name)
...     loader = dc.data.CSVLoader(["task1"], feature_field="smiles", featurizer=dc.
↪feat.DummyFeaturizer())
...     dataset = loader.create_dataset(tmpfile.name)
```

```
>>> # pretraining
>>> from deepchem.models.torch_models.hf_models import HuggingFaceModel
>>> from transformers.models.roberta import RobertaForMaskedLM, RobertaModel,␣
↪RobertaConfig
>>> config = RobertaConfig(vocab_size=tokenizer.vocab_size)
>>> model = RobertaForMaskedLM(config)
```

(continues on next page)

(continued from previous page)

```
>>> hf_model = HuggingFaceModel(model=model, tokenizer=tokenizer, task='mlm', model_
↪dir='model-dir')
>>> training_loss = hf_model.fit(dataset, nb_epoch=1)
```

```
>>> # finetuning a regression model
>>> from transformers.models.roberta import RobertaForSequenceClassification
>>> config = RobertaConfig(vocab_size=tokenizer.vocab_size, problem_type='regression
↪', num_labels=1)
>>> model = RobertaForSequenceClassification(config)
>>> hf_model = HuggingFaceModel(model=model, tokenizer=tokenizer, task='regression',
↪ model_dir='model-dir')
>>> hf_model.load_from_pretrained()
>>> training_loss = hf_model.fit(dataset, nb_epoch=1)
>>> prediction = hf_model.predict(dataset)  # prediction
>>> eval_results = hf_model.evaluate(dataset, metrics=dc.metrics.Metric(dc.metrics.
↪mae_score))
```

```
>>> # finetune a classification model
>>> # making dataset suitable for classification
>>> import numpy as np
>>> y = np.random.choice([0, 1], size=dataset.y.shape)
>>> dataset = dc.data.NumpyDataset(X=dataset.X, y=y, w=dataset.w, ids=dataset.ids)
```

```
>>> from transformers import RobertaForSequenceClassification
>>> config = RobertaConfig(vocab_size=tokenizer.vocab_size)
>>> model = RobertaForSequenceClassification(config)
>>> hf_model = HuggingFaceModel(model=model, task='classification',␣
↪tokenizer=tokenizer)
>>> training_loss = hf_model.fit(dataset, nb_epoch=1)
>>> predictions = hf_model.predict(dataset)
>>> eval_result = hf_model.evaluate(dataset, metrics=dc.metrics.Metric(dc.metrics.
↪f1_score))
```

`__init__`(*model: PreTrainedModel*, *tokenizer: transformers.tokenization_utils.PreTrainedTokenizer*, *task: str | None = None*, ***kwargs*)

Create a new TorchModel.

Parameters

- **model** (*torch.nn.Module*) – the PyTorch model implementing the calculation
- **loss** (*dc.models.losses.Loss or function*) – a Loss or function defining how to compute the training loss for each batch, as described above
- **output_types** (*list of strings, optional (default None)*) – the type of each output from the model, as described above
- **batch_size** (*int, optional (default 100)*) – default batch size for training and evaluating
- **model_dir** (*str, optional (default None)*) – the directory on disk where the model will be stored. If this is None, a temporary directory is created.
- **learning_rate** (*float or LearningRateSchedule, optional (default 0.001)*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.

- **optimizer** (Optimizer, *optional (default None)*) – the optimizer to use for fitting. If this is specified, learning_rate is ignored.
- **tensorboard** (*bool, optional (default False)*) – whether to log progress to TensorBoard during training
- **wandb** (*bool, optional (default False)*) – whether to log progress to Weights & Biases during training
- **log_frequency** (*int, optional (default 100)*) – The frequency at which to log data. Data is logged using *logging* by default. If *tensorboard* is set, data is also logged to TensorBoard. If *wandb* is set, data is also logged to Weights & Biases. Logging happens at global steps. Roughly, a global step corresponds to one batch of training. If you'd like a printout every 10 batch steps, you'd set *log_frequency=10* for example.
- **device** (*torch.device, optional (default None)*) – the device on which to run computations. If None, a device is chosen automatically.
- **regularization_loss** (*Callable, optional*) – a function that takes no arguments, and returns an extra contribution to add to the loss function
- **wandb_logger** (*WandbLogger*) – the Weights & Biases logger object used to log data and metrics

load_from_pretrained(*model_dir: str | None = None*, *from_hf_checkpoint: bool = False*)

Load HuggingFace model from a pretrained checkpoint.

The utility can be used for loading a model from a checkpoint. Given *model_dir*, it checks for existing checkpoint in the directory. If a checkpoint exists, the models state is loaded from the checkpoint.

If the option *from_hf_checkpoint* is set as True, then it loads a pretrained model using HuggingFace models *from_pretrained* method. This option interprets model_dir as a model id of a pretrained model hosted inside a model repo on huggingface.co or path to directory containing model weights saved using *save_pretrained* method of a HuggingFace model.

3.21.1 Parameter

model_dir: str
Directory containing model checkpoint

from_hf_checkpoint: bool, default False
Loads a pretrained model from HuggingFace checkpoint.

Example

```
>>> from transformers import RobertaTokenizerFast
>>> tokenizer = RobertaTokenizerFast.from_pretrained("seyonec/PubChem10M_SMILES_
↪BPE_60k")
```

```
>>> from deepchem.models.torch_models.hf_models import HuggingFaceModel
>>> from transformers.models.roberta import RobertaForMaskedLM, RobertaModel,␣
↪RobertaConfig
>>> config = RobertaConfig(vocab_size=tokenizer.vocab_size)
>>> model = RobertaForMaskedLM(config)
>>> pretrain_model = HuggingFaceModel(model=model, tokenizer=tokenizer, task=
```

(continues on next page)

(continued from previous page)

```
↪'mlm', model_dir='model-dir')
>>> pretrain_model.save_checkpoint()
```

```
>>> from transformers import RobertaForSequenceClassification
>>> config = RobertaConfig(vocab_size=tokenizer.vocab_size)
>>> model = RobertaForSequenceClassification(config)
>>> finetune_model = HuggingFaceModel(model=model, task='classification',␣
↪tokenizer=tokenizer, model_dir='model-dir')
```

```
>>> finetune_model.load_from_pretrained()
```

fit_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *restore: bool = False*, *variables: List[Parameter] | ParameterList | None = None*, *loss: Callable[[List, List, List], Any] | None = None*, *callbacks: Callable | List[Callable] = []*, *all_losses: List[float] | None = None*) → float

Train this model on data from a generator.

Parameters

- **generator** (*generator*) – this should generate batches, each represented as a tuple of the form (inputs, labels, weights).
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **checkpoint_interval** (*int*) – the frequency at which to write checkpoints, measured in training steps. Set this to 0 to disable automatic checkpointing.
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.
- **variables** (*list of torch.nn.Parameter*) – the variables to train. If None (the default), all trainable variables in the model are used.
- **loss** (*function*) – a function of the form f(outputs, labels, weights) that computes the loss for each batch. If None (the default), the model's standard loss function is used.
- **callbacks** (*function or list of functions*) – one or more functions of the form f(model, step, **kwargs) that will be invoked after every step. This can be used to perform validation, logging, etc.
- **all_losses** (*Optional[List[float]], optional (default None)*) – If specified, all logged losses are appended into this list. Note that you can call *fit()* repeatedly with the same list and losses will continue to be appended.

Return type

The average loss over the most recent checkpoint interval

Note: A HuggingFace model can return embeddings (last hidden state), attentions. Support must be added to return the embeddings to the user, so that it can be used for other downstream applications.

3.21.2 Chemberta

class Chemberta(*task: str*, *tokenizer_path: str = 'seyonec/PubChem10M_SMILES_BPE_60k'*, *n_tasks: int = 1*, *config: Dict[Any, Any] = {}*, ***kwargs*)

Chemberta Model

Chemberta is a transformer style model for learning on SMILES strings. The model architecture is based on the RoBERTa architecture. The model has can be used for both pretraining an embedding and finetuning for downstream applications.

The model supports two types of pretraining tasks - pretraining via masked language modeling and pretraining via multi-task regression. To pretrain via masked language modeling task, use task = *mlm* and for pretraining via multitask regression task, use task = *mtr*. The model supports the regression, classification and multitask regression finetuning tasks and they can be specified using *regression*, *classification* and *mtr* as arguments to the *task* keyword during model initialisation.

The model uses a tokenizer To create input tokens for the models from the SMILES strings. The default tokenizer model is a byte-pair encoding tokenizer trained on PubChem10M dataset and loaded from huggingFace model hub (https://huggingface.co/seyonec/PubChem10M_SMILES_BPE_60k).

Parameters

- **task** (*str*) –

  The task defines the type of learning task in the model. The supported tasks are

  – *mlm* - masked language modeling commonly used in pretraining

  – *mtr* - multitask regression - a task used for both pretraining base models and finetuning

  – *regression* - use it for regression tasks, like property prediction

  – *classification* - use it for classification tasks

- **tokenizer_path** (*str*) – Path containing pretrained tokenizer used to tokenize SMILES string for model inputs. The tokenizer path can either be a huggingFace tokenizer model or a path in the local machine containing the tokenizer.

- **n_tasks** (*int, default 1*) – Number of prediction targets for a multitask learning model

Example

```
>>> import os
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
```

```
>>> # preparing dataset
>>> import pandas as pd
>>> import deepchem as dc
>>> smiles = ["CCN(CCSC)C(=O)N[C@@](C)(CC)C(F)(F)F",
↪"CC1(C)CN(C(=O)Nc2cc3ccccc3nn2)C[C@@]2(CCOC2)O1"]
>>> labels = [3.112,2.432]
>>> df = pd.DataFrame(list(zip(smiles, labels)), columns=["smiles", "task1"])
>>> with dc.utils.UniversalNamedTemporaryFile(mode='w') as tmpfile:
...     df.to_csv(tmpfile.name)
...     loader = dc.data.CSVLoader(["task1"], feature_field="smiles", featurizer=dc.
↪feat.DummyFeaturizer())
...     dataset = loader.create_dataset(tmpfile.name)
```

```
>>> # pretraining
>>> from deepchem.models.torch_models.chemberta import Chemberta
>>> pretrain_model_dir = os.path.join(tempdir, 'pretrain-model')
>>> tokenizer_path = "seyonec/PubChem10M_SMILES_BPE_60k"
>>> pretrain_model = Chemberta(task='mlm', model_dir=pretrain_model_dir, tokenizer_
↪path=tokenizer_path)  # mlm pretraining
>>> pretraining_loss = pretrain_model.fit(dataset, nb_epoch=1)
```

```
>>> # finetuning in regression mode
>>> finetune_model_dir = os.path.join(tempdir, 'finetune-model')
>>> finetune_model = Chemberta(task='regression', model_dir=finetune_model_dir,␣
↪tokenizer_path=tokenizer_path)
>>> finetune_model.load_from_pretrained(pretrain_model_dir)
>>> finetuning_loss = finetune_model.fit(dataset, nb_epoch=1)
```

```
>>> # prediction and evaluation
>>> result = finetune_model.predict(dataset)
>>> eval_results = finetune_model.evaluate(dataset, metrics=dc.metrics.Metric(dc.
↪metrics.mae_score))
```

Reference

__init__(*task: str*, *tokenizer_path: str = 'seyonec/PubChem10M_SMILES_BPE_60k'*, *n_tasks: int = 1*, *config: Dict[Any, Any] = {}*, ***kwargs*)

Create a new TorchModel.

Parameters

- **model** (*torch.nn.Module*) – the PyTorch model implementing the calculation
- **loss** (*dc.models.losses.Loss or function*) – a Loss or function defining how to compute the training loss for each batch, as described above
- **output_types** (*list of strings, optional (default None)*) – the type of each output from the model, as described above
- **batch_size** (*int, optional (default 100)*) – default batch size for training and evaluating
- **model_dir** (*str, optional (default None)*) – the directory on disk where the model will be stored. If this is None, a temporary directory is created.
- **learning_rate** (*float or LearningRateSchedule, optional (default 0.001)*) – the learning rate to use for fitting. If optimizer is specified, this is ignored.
- **optimizer** (*Optimizer, optional (default None)*) – the optimizer to use for fitting. If this is specified, learning_rate is ignored.
- **tensorboard** (*bool, optional (default False)*) – whether to log progress to TensorBoard during training
- **wandb** (*bool, optional (default False)*) – whether to log progress to Weights & Biases during training
- **log_frequency** (*int, optional (default 100)*) – The frequency at which to log data. Data is logged using *logging* by default. If *tensorboard* is set, data is also logged to TensorBoard. If *wandb* is set, data is also logged to Weights & Biases. Logging happens

at global steps. Roughly, a global step corresponds to one batch of training. If you'd like a printout every 10 batch steps, you'd set *log_frequency=10* for example.

- **device** (*torch.device, optional (default None)*) – the device on which to run computations. If None, a device is chosen automatically.
- **regularization_loss** (*Callable, optional*) – a function that takes no arguments, and returns an extra contribution to add to the loss function
- **wandb_logger** (*WandbLogger*) – the Weights & Biases logger object used to log data and metrics

3.22 Trainer

A *Trainer* object automates the scaling of DeepChem model's training into multi-gpu and multi-node infrastructures.

3.22.1 DistributedTrainer

3.23 Layers

Deep learning models are often said to be made up of "layers". Intuitively, a "layer" is a function which transforms some tensor into another tensor. DeepChem maintains an extensive collection of layers which perform various useful scientific transformations. For now, most layers are Keras only but over time we expect this support to expand to other types of models and layers.

3.23.1 Layers Cheatsheet

The "layers cheatsheet" lists various scientifically relevant differentiable layers implemented in DeepChem.

Note that some layers implemented for specific model architectures such as `GROVER` and `Attention` layers, this is indicated in the *Model* column of the table.

In order to use the layers, make sure that the backend (Keras and tensorflow, Pytorch or Jax) is installed.

Tensorflow Keras Layers

These layers are subclasses of the `tensorflow.keras.layers.Layer` class.

Table 7: Custom Keras Layers

Layer	Reference	Model
InteratomicL2Distances		
GraphConv	ref	
GraphPool	ref	
GraphGather	ref	
MolGANConvolutionLayer	ref	MolGan
MolGANAggregationLayer	ref	MolGan
MolGANMultiConvolutionLayer	ref	MolGan
MolGANEncoderLayer	ref	MolGan
LSTMStep		
AttnLSTMEmbedding	ref	
IterRefLSTMEmbedding		

continues on next page

Table 7 – continued from previous page

Layer	Reference	Model
SwitchedDropout		
WeightedLinearCombo		
CombineMeanSt		
Stack		
VinaFreeEnergy		
NeighborList		
AtomicConvolution	ref	
AlphaShareLayer		Sluice Network
SluiceLoss		Sluice Network
BetaShare		Sluice Network
ANIFeat		
GraphEmbedPoolLayer	ref	
Highway	ref	
WeaveLayer	ref	
WeaveGather	ref	
DTNNEmbedding		
DTNNStep		
DTNNGather		
DAGLayer	ref	
DAGGather		
MessagePassing	ref	
EdgeNetwork	ref	MessagePassing
GatedRecurrentUnit	ref	MessagePassing
SetGather		

PyTorch

These layers are subclasses of the `torch.nn.Module` class.

Table 8: Custom PyTorch Layers

Layer	Reference	Model
MultilayerPerceptron		
ScaleNorm	ref	Molecular Attention Transformer
MATEncoderLayer	ref	Molecular Attention Transformer
MultiHeadedMATAttention	ref	Molecular Attention Transformer
SublayerConnection	ref	Transformer
MATEmbedding	ref	Molecular Attention Transformer
MATGenerator	ref	Molecular Attention Transformer
Affine	ref	Normalizing Flow
RealNVPLayer	ref	Normalizing Flow
DMPNNEncoderLayer	ref	Normalizing Flow
PositionwiseFeedForward	ref	Molecular Attention Transformer
GraphPool	ref	
GroverMPNEncoder	ref	Grover
GroverAttentionHead	ref	Grover
GroverMTBlock	ref	Grover
GroverTransEncoder	ref	Grover
GroverEmbedding	ref	Grover
GroverAtomVocabPredictor	ref	Grover

continues on next page

Table 8 – continued from previous page

Layer	Reference	Model
GroverBondVocabPredictor	ref	Grover
GroverFunctionalGroupPredictor	ref	Grover
ScaledDotProductAttention	ref	Transformer
SelfAttention	ref	Transformer
GroverReadout	ref	Grover
DFTXC	ref	XCModel-DFT
NNLDA	ref	XCModel-DFT
HybridXC	ref	XCModel-DFT
XCNNSCF	ref	XCModel-DFT
AtomEncoder	**`https://arxiv.org/abs/2110.04126`_**	3D InfoMax
BondEncoder	**`https://arxiv.org/abs/2110.04126`_**	3D InfoMax
Net3DLayer	**`https://arxiv.org/abs/2110.04126`_**	3D InfoMax
Net3D	**`https://arxiv.org/abs/2110.04126`_**	3D InfoMax
PNALayer	**`https://arxiv.org/abs/2004.05718`_**	Principal Neighbourhood Aggregation
PNAGNN	**`https://arxiv.org/abs/2004.05718`_**	Principal Neighbourhood Aggregation
EdgeNetwork	ref	Message Passing Neural Network
WeaveLayer	ref	WeaveModel
WeaveGather	ref	WeaveModel
GradientPenalty	ref	WGANModel
MolGANConvolutionLayer	ref	MolGan
MolGANAggregationLayer	ref	MolGan
MolGANMultiConvolutionLayer	ref	MolGan
MolGANEncoderLayer	ref	MolGan
DTNNEmbedding	ref`<https://arxiv.org/abs/1609.08259>`_	DTNNModel
DTNNStep	ref`<https://arxiv.org/abs/1609.08259>`_	DTNNModel
DTNNGather	ref`<https://arxiv.org/abs/1609.08259>`_	DTNNModel
MXMNetGlobalMessagePassing	ref	MXMNetModel
MXMNetBesselBasisLayer	ref	MXMNetModel
VariationalRandomizer	ref	SeqToSeqModel
EncoderRNN	ref	SeqToSeqModel
DecoderRNN	ref	SeqToSeqModel
FerminetElectronFeature	ref	FerminetModel
FerminetEnvelope	ref	FerminetModel
MXMNetLocalMessagePassing	ref	MXMNetModel
MXMNetModelMXMNetSphericalBasisLayer	ref`<https://arxiv.org/pdf/2011.07457>`_	MXMNetModel
HighwayLayer	ref	

3.23.2 Keras Layers

class InteratomicL2Distances(**args*, ***kwargs*)

Compute (squared) L2 Distances between atoms given neighbors.

This class computes pairwise distances between its inputs.

Examples

```
>>> import numpy as np
>>> import deepchem as dc
>>> atoms = 5
>>> neighbors = 2
>>> coords = np.random.rand(atoms, 3)
>>> neighbor_list = np.random.randint(0, atoms, size=(atoms, neighbors))
>>> layer = InteratomicL2Distances(atoms, neighbors, 3)
>>> result = np.array(layer([coords, neighbor_list]))
>>> result.shape
(5, 2)
```

`__init__`(*N_atoms: int*, *M_nbrs: int*, *ndim: int*, ***kwargs*)

Constructor for this layer.

Parameters

- **`N_atoms`** (*`int`*) – Number of atoms in the system total.
- **`M_nbrs`** (*`int`*) – Number of neighbors to consider when computing distances.
- **`n_dim`** (*`int`*) – Number of descriptors for each atom.

`get_config`() → Dict

Returns config dictionary for this layer.

`call`(*inputs: List*)

Invokes this layer.

Parameters

`inputs` (*`list`*) – Should be of form *inputs=[coords, nbr_list]* where *coords* is a tensor of shape *(None, N, 3)* and *nbr_list* is a list.

Return type

Tensor of shape *(N_atoms, M_nbrs)* with interatomic distances.

class GraphConv(**args*, ***kwargs*)

Graph Convolutional Layers

This layer implements the graph convolution introduced in [1]_. The graph convolution combines per-node feature vectures in a nonlinear fashion with the feature vectors for neighboring nodes. This "blends" information in local neighborhoods of a graph.

References

`__init__`(*out_channel: int*, *min_deg: int = 0*, *max_deg: int = 10*, *activation_fn: Callable | None = None*, ***kwargs*)

Initialize a graph convolutional layer.

Parameters

- **`out_channel`** (*`int`*) – The number of output channels per graph node.
- **`min_deg`** (*`int, optional (default 0)`*) – The minimum allowed degree for each graph node.

- `max_deg` (*int, optional (default 10)*) – The maximum allowed degree for each graph node. Note that this is set to 10 to handle complex molecules (some organometallic compounds have strange structures). If you're using this for non-molecular applications, you may need to set this much higher depending on your dataset.
- `activation_fn` (*function*) – A nonlinear activation function to apply. If you're not sure, *tf.nn.relu* is probably a good default for your application.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters

`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns

Python dictionary.

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.

  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns

A tensor or list/tuple of tensors.

`sum_neigh`(*atoms*, *deg_adj_lists*)

Store the summed atoms by degree

`class GraphPool`(**args*, ***kwargs*)

A GraphPool gathers data from local neighborhoods of a graph.

This layer does a max-pooling over the feature vectors of atoms in a neighborhood. You can think of this layer as analogous to a max-pooling layer for 2D convolutions but which operates on graphs instead. This technique is described in [1]_.

References

`__init__`(*min_degree=0*, *max_degree=10*, ***kwargs*)

Initialize this layer

Parameters

- **min_deg** (*int, optional (default 0)*) – The minimum allowed degree for each graph node.
- **max_deg** (*int, optional (default 10)*) – The maximum allowed degree for each graph node. Note that this is set to 10 to handle complex molecules (some organometallic compounds have strange structures). If you're using this for non-molecular applications, you may need to set this much higher depending on your dataset.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **`inputs`** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.
  – Keras mask metadata is only collected from *inputs*.
  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  – *input_spec* compatibility is only checked against *inputs*.
  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  – The SavedModel input specification is generated using *inputs* only.
  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **`*args`** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **`**kwargs`** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

class **GraphGather**(**args*, ***kwargs*)

A GraphGather layer pools node-level feature vectors to create a graph feature vector.

Many graph convolutional networks manipulate feature vectors per graph-node. For a molecule for example, each node might represent an atom, and the network would manipulate atomic feature vectors that summarize the local chemistry of the atom. However, at the end of the application, we will likely want to work with a molecule level feature representation. The *GraphGather* layer creates a graph level feature vector by combining all the node-level feature vectors.

One subtlety about this layer is that it depends on the *batch_size*. This is done for internal implementation reasons. The *GraphConv*, and *GraphPool* layers pool all nodes from all graphs in a batch that's being processed. The *GraphGather* reassembles these jumbled node feature vectors into per-graph feature vectors.

References

__init__(*batch_size*, *activation_fn=None*, ***kwargs*)

Initialize this layer.

Parameters

- **batch_size** (*int*) – The batch size for this layer. Note that the layer's behavior changes depending on the batch size.
- **activation_fn** (*function*) – A nonlinear activation function to apply. If you're not sure, *tf.nn.relu* is probably a good default for your application.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns

Python dictionary.

call(*inputs*)

Invoking this layer.

Parameters

inputs (*list*) – This list should consist of *inputs = [atom_features, deg_slice, membership, deg_adj_list placeholders. . .]*. These are all tensors that are created/process by *GraphConv* and *GraphPool*

class **MolGANConvolutionLayer**(**args*, ***kwargs*)

Graph convolution layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. Not used directly, higher level layers like MolGANMultiConvolutionLayer use it. This layer performs basic convolution on one-hot encoded matrices containing atom and bond information. This layer also accepts three inputs for the case when convolution is performed more than once and results of previous convolution need to used. It was done in such a way to avoid creating another layer that accepts three inputs rather than two. The last input layer is so-called hidden_layer and it hold results of the convolution while first two are unchanged input tensors.

Example

See: MolGANMultiConvolutionLayer for using in layers.

```
>>> from tensorflow.keras import Model
>>> from tensorflow.keras.layers import Input
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = 128
```

```
>>> layer1 = MolGANConvolutionLayer(units=units,edges=edges, name='layer1')
>>> layer2 = MolGANConvolutionLayer(units=units,edges=edges, name='layer2')
>>> adjacency_tensor= Input(shape=(vertices, vertices, edges))
>>> node_tensor = Input(shape=(vertices,nodes))
>>> hidden1 = layer1([adjacency_tensor,node_tensor])
>>> output = layer2(hidden1)
>>> model = Model(inputs=[adjacency_tensor,node_tensor], outputs=[output])
```

References

`__init__`(*units: int*, *activation: ~typing.Callable = <function tanh>*, *dropout_rate: float = 0.0*, *edges: int = 5*, *name: str = ''*, ***kwargs*)

Initialize this layer.

Parameters

- **units** (*int*) – Dimesion of dense layers used for convolution
- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Dropout rate used by dropout layer
- **edges** (*int, optional (default=5)*) – How many dense layers to use in convolution. Typically equal to number of bond types used in the model.
- **name** (*string, optional (default="")*) – Name of the layer

`call`(*inputs*, *training=False*)

Invoke this layer

Parameters

- **inputs** (*list*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.
- **training** (*bool*) – Should this layer be run in training mode. Typically decided by main model, influences things like dropout.

Returns

First and second are original input tensors Third is the result of convolution

Return type

tuple(tf.Tensor,tf.Tensor,tf.Tensor)

get_config() → Dict

Returns config dictionary for this layer.

class **MolGANAggregationLayer**(**args*, ***kwargs*)

Graph Aggregation layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. Performs aggregation on tensor resulting from convolution layers. Given its simple nature it might be removed in future and moved to MolGANEncoderLayer.

Example

```
>>> from tensorflow.keras import Model
>>> from tensorflow.keras.layers import Input
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = 128
```

```
>>> layer_1 = MolGANConvolutionLayer(units=units,edges=edges, name='layer1')
>>> layer_2 = MolGANConvolutionLayer(units=units,edges=edges, name='layer2')
>>> layer_3 = MolGANAggregationLayer(units=128, name='layer3')
>>> adjacency_tensor= Input(shape=(vertices, vertices, edges))
>>> node_tensor = Input(shape=(vertices,nodes))
>>> hidden_1 = layer_1([adjacency_tensor,node_tensor])
>>> hidden_2 = layer_2(hidden_1)
>>> output = layer_3(hidden_2[2])
>>> model = Model(inputs=[adjacency_tensor,node_tensor], outputs=[output])
```

References

__init__(*units: int = 128*, *activation: ~typing.Callable = <function tanh>*, *dropout_rate: float = 0.0*, *name: str = ''*, ***kwargs*)

Initialize the layer

Parameters

- **units** (*int, optional (default=128)*) – Dimesion of dense layers used for aggregation
- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Used by dropout layer
- **name** (*string, optional (default="")*) – Name of the layer

call(*inputs*, *training=False*)

Invoke this layer

Parameters

- **inputs** (*List*) – Single tensor resulting from graph convolution layer
- **training** (*bool*) – Should this layer be run in training mode. Typically decided by main model, influences things like dropout.

Returns
aggregation tensor – Result of aggregation function on input convolution tensor.

Return type
tf.Tensor

get_config() → Dict

Returns config dictionary for this layer.

class MolGANMultiConvolutionLayer(**args*, ***kwargs*)

Multiple pass convolution layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. It takes outputs of previous convolution layer and uses them as inputs for the next one. It simplifies the overall framework, but might be moved to MolGANEncoderLayer in the future in order to reduce number of layers.

Example

```
>>> from tensorflow.keras import Model
>>> from tensorflow.keras.layers import Input
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = 128
```

```
>>> layer_1 = MolGANMultiConvolutionLayer(units=(128,64), name='layer1')
>>> layer_2 = MolGANAggregationLayer(units=128, name='layer2')
>>> adjacency_tensor= Input(shape=(vertices, vertices, edges))
>>> node_tensor = Input(shape=(vertices,nodes))
>>> hidden = layer_1([adjacency_tensor,node_tensor])
>>> output = layer_2(hidden)
>>> model = Model(inputs=[adjacency_tensor,node_tensor], outputs=[output])
```

References

__init__(*units: ~typing.Tuple = (128, 64), activation: ~typing.Callable = <function tanh>, dropout_rate: float = 0.0, edges: int = 5, name: str = '', **kwargs*)

Initialize the layer

Parameters

- **units** (*Tuple, optional (default=(128,64)), min_length=2*) – List of dimensions used by consecutive convolution layers. The more values the more convolution layers invoked.
- **activation** (*function, optional (default=tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Used by dropout layer
- **edges** (*int, optional (default=0)*) – Controls how many dense layers use for single convolution unit. Typically matches number of bond types used in the molecule.
- **name** (*string, optional (default="")*) – Name of the layer

call(*inputs*, *training=False*)

Invoke this layer

Parameters

- **inputs** (*list*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.
- **training** (*bool*) – Should this layer be run in training mode. Typically decided by main model, influences things like dropout.

Returns

convolution tensor – Result of input tensors going through convolution a number of times.

Return type

tf.Tensor

get_config() → Dict

Returns config dictionary for this layer.

class MolGANEncoderLayer(**args*, ***kwargs*)

Main learning layer used by MolGAN model. MolGAN is a WGAN type model for generation of small molecules. It role is to further simplify model. This layer can be manually built by stacking graph convolution layers followed by graph aggregation.

Example

```
>>> from tensorflow.keras import Model
>>> from tensorflow.keras.layers import Input, Dropout,Dense
>>> vertices = 9
>>> edges = 5
>>> nodes = 5
>>> dropout_rate = .0
>>> adjacency_tensor= Input(shape=(vertices, vertices, edges))
>>> node_tensor = Input(shape=(vertices, nodes))
```

```
>>> graph = MolGANEncoderLayer(units = [(128,64),128], dropout_rate= dropout_rate,␣
↪edges=edges)([adjacency_tensor,node_tensor])
>>> dense = Dense(units=128, activation='tanh')(graph)
>>> dense = Dropout(dropout_rate)(dense)
>>> dense = Dense(units=64, activation='tanh')(dense)
>>> dense = Dropout(dropout_rate)(dense)
>>> output = Dense(units=1)(dense)
>>> model = Model(inputs=[adjacency_tensor,node_tensor], outputs=[output])
```

References

`__init__`(*units: ~typing.List = [(128, 64), 128], activation: ~typing.Callable = <function tanh>, dropout_rate: float = 0.0, edges: int = 5, name: str = '', **kwargs*)

Initialize the layer.

Parameters

- **`units`** (*`List, optional (default=[(128, 64), 128])`*) – List of units for MolGANMultiConvolutionLayer and GraphAggregationLayer i.e. [(128,64),128] means two convolution layers dims = [128,64] followed by aggregation layer dims=128
- **`activation`** (*`function, optional (default=Tanh)`*) – activation function used across model, default is Tanh
- **`dropout_rate`** (*`float, optional (default=0.0)`*) – Used by dropout layer
- **`edges`** (*`int, optional (default=0)`*) – Controls how many dense layers use for single convolution unit. Typically matches number of bond types used in the molecule.
- **`name`** (*`string, optional (default="")`*) – Name of the layer

`call`(*inputs*, *training=False*)

Invoke this layer

Parameters

- **`inputs`** (*`list`*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.
- **`training`** (*`bool`*) – Should this layer be run in training mode. Typically decided by main model, influences things like dropout.

Returns

encoder tensor – Tensor that been through number of convolutions followed by aggregation.

Return type

tf.Tensor

`get_config`() → Dict

Returns config dictionary for this layer.

class `LSTMStep`(**args*, ***kwargs*)

Layer that performs a single step LSTM update.

This layer performs a single step LSTM update. Note that it is *not* a full LSTM recurrent network. The LSTMStep layer is useful as a primitive for designing layers such as the AttnLSTMEmbedding or the IterRefLSTMEmbedding below.

`__init__`(*output_dim*, *input_dim*, *init_fn='glorot_uniform'*, *inner_init_fn='orthogonal'*, *activation_fn='tanh'*, *inner_activation_fn='hard_sigmoid'*, ***kwargs*)

Parameters

- **`output_dim`** (*`int`*) – Dimensionality of output vectors.
- **`input_dim`** (*`int`*) – Dimensionality of input vectors.
- **`init_fn`** (*`str`*) – TensorFlow nitialization to use for W.

- **inner_init_fn** (*str*) – TensorFlow initialization to use for U.
- **activation_fn** (*str*) – TensorFlow activation to use for output.
- **inner_activation_fn** (*str*) – TensorFlow activation to use for inner steps.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Constructs learnable weights for this layer.

call(*inputs*)

Execute this layer on input tensors.

Parameters
inputs (*list*) – List of three tensors (x, h_tm1, c_tm1). h_tm1 means "h, t-1".

Returns
Returns h, [h, c]

Return type
list

class **AttnLSTMEmbedding**(**args*, ***kwargs*)

Implements AttnLSTM as in matching networks paper.

The AttnLSTM embedding adjusts two sets of vectors, the "test" and "support" sets. The "support" consists of a set of evidence vectors. Think of these as the small training set for low-data machine learning. The "test" consists of the queries we wish to answer with the small amounts of available data. The AttnLSTMEmbdding allows us to modify the embedding of the "test" set depending on the contents of the "support". The AttnLSTMEmbedding is thus a type of learnable metric that allows a network to modify its internal notion of distance.

See references [1]_ [2]_ for more details.

References

__init__(*n_test*, *n_support*, *n_feat*, *max_depth*, ***kwargs*)

Parameters

- **n_support** (*int*) – Size of support set.
- **n_test** (*int*) – Size of test set.
- **n_feat** (*int*) – Number of features per atom
- **max_depth** (*int*) – Number of "processing steps" used by sequence-to-sequence for sets model.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*)

Execute this layer on input tensors.

Parameters
inputs (`list`) – List of two tensors (X, Xp). X should be of shape (n_test, n_feat) and Xp should be of shape (n_support, n_feat) where n_test is the size of the test set, n_support that of the support set, and n_feat is the number of per-atom features.

Returns
Returns two tensors of same shape as input. Namely the output shape will be [(n_test, n_feat), (n_support, n_feat)]

Return type
list

class **IterRefLSTMEmbedding**(**args*, ***kwargs*)

Implements the Iterative Refinement LSTM.

Much like AttnLSTMEmbedding, the IterRefLSTMEmbedding is another type of learnable metric which adjusts "test" and "support." Recall that "support" is the small amount of data available in a low data machine learning problem, and that "test" is the query. The AttnLSTMEmbedding only modifies the "test" based on the contents of the support. However, the IterRefLSTM modifies both the "support" and "test" based on each other. This allows the learnable metric to be more malleable than that from AttnLSTMEmbeding.

__init__(*n_test*, *n_support*, *n_feat*, *max_depth*, ***kwargs*)

Unlike the AttnLSTM model which only modifies the test vectors additively, this model allows for an additive update to be performed to both test and support using information from each other.

Parameters

- **n_support** (`int`) – Size of support set.
- **n_test** (`int`) – Size of test set.

- `n_feat` (*int*) – Number of input atom features
- `max_depth` (*int*) – Number of LSTM Embedding layers.

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

Execute this layer on input tensors.

Parameters
`inputs` (*list*) – List of two tensors (X, Xp). X should be of shape (n_test, n_feat) and Xp should be of shape (n_support, n_feat) where n_test is the size of the test set, n_support that of the support set, and n_feat is the number of per-atom features.

Returns

- *Returns two tensors of same shape as input. Namely the output*
- *shape will be [(n_test, n_feat), (n_support, n_feat)]*

`class SwitchedDropout`(**args*, ***kwargs*)

Apply dropout based on an input.

This is required for uncertainty prediction. The standard Keras Dropout layer only performs dropout during training, but we sometimes need to do it during prediction. The second input to this layer should be a scalar equal to 0 or 1, indicating whether to perform dropout.

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.
- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class WeightedLinearCombo`(**args*, ***kwargs*)

Computes a weighted linear combination of input layers, with the weights defined by trainable variables.

`__init__`(*std=0.3*, ***kwargs*)

Initialize this layer.

Parameters

`std` (*float, optional (default 0.3)*) – The standard deviation to use when randomly initializing weights.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns

Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters

`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **`*args`** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **`**kwargs`** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class CombineMeanStd(`**args*, ***kwargs*`)`

Generate Gaussian nose.

`__init__(`*training_only=False*, *noise_epsilon=1.0*, ***kwargs*`)`

Create a CombineMeanStd layer.

This layer should have two inputs with the same shape, and its output also has the same shape. Each element of the output is a Gaussian distributed random number whose mean is the corresponding element of the first input, and whose standard deviation is the corresponding element of the second input.

Parameters

- **`training_only`** (*`bool`*) – if True, noise is only generated during training. During prediction, the output is simply equal to the first input (that is, the mean of the distribution used during training).
- **`noise_epsilon`** (*`float`*) – The noise is scaled by this factor

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*, *training=True*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- `*args` – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- `**kwargs` – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns

A tensor or list/tuple of tensors.

`class Stack`(**args*, ***kwargs*)

Stack the inputs along a new axis.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **`inputs`** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.

  – The SavedModel input specification is generated using *inputs* only.

  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **`*args`** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **`**kwargs`** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

class **VinaFreeEnergy**(**args*, ***kwargs*)

Computes free-energy as defined by Autodock Vina.

TODO(rbharath): Make this layer support batching.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

nonlinearity(*c*, *w*)

Computes non-linearity used in Vina.

repulsion(*d*)

Computes Autodock Vina's repulsion interaction term.

hydrophobic(*d*)

Computes Autodock Vina's hydrophobic interaction term.

hydrogen_bond(*d*)

Computes Autodock Vina's hydrogen bond interaction term.

gaussian_first(*d*)

Computes Autodock Vina's first Gaussian interaction term.

gaussian_second(*d*)

Computes Autodock Vina's second Gaussian interaction term.

call(*inputs*)

Parameters

- **X** (*tf.Tensor of shape (N, d)*) – Coordinates/features.
- **Z** (*tf.Tensor of shape (N)*) – Atomic numbers of neighbor atoms.

Returns
layer – The free energy of each complex in batch

Return type
tf.Tensor of shape (B)

`class NeighborList(*args, **kwargs)`

Computes a neighbor-list in Tensorflow.

Neighbor-lists (also called Verlet Lists) are a tool for grouping atoms which are close to each other spatially. This layer computes a Neighbor List from a provided tensor of atomic coordinates. You can think of this as a general "k-means" layer, but optimized for the case *k==3*.

TODO(rbharath): Make this layer support batching.

`__init__(N_atoms, M_nbrs, ndim, nbr_cutoff, start, stop, **kwargs)`

Parameters

- **N_atoms** (*int*) – Maximum number of atoms this layer will neighbor-list.
- **M_nbrs** (*int*) – Maximum number of spatial neighbors possible for atom.
- **ndim** (*int*) – Dimensionality of space atoms live in. (Typically 3D, but sometimes will want to use higher dimensional descriptors for atoms).
- **nbr_cutoff** (*float*) – Length in Angstroms (?) at which atom boxes are gridded.

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call(inputs)`

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`compute_nbr_list`(*coords*)

Get closest neighbors for atoms.

Needs to handle padding for atoms with no neighbors.

Parameters
coords (*tf.Tensor*) – Shape (N_atoms, ndim)

Returns
nbr_list – Shape (N_atoms, M_nbrs) of atom indices

Return type
tf.Tensor

`get_atoms_in_nbrs`(*coords*, *cells*)

Get the atoms in neighboring cells for each cells.

Return type
atoms_in_nbrs = (N_atoms, n_nbr_cells, M_nbrs)

`get_closest_atoms`(*coords*, *cells*)

For each cell, find M_nbrs closest atoms.

Let N_atoms be the number of atoms.

Parameters

- **coords** (*tf.Tensor*) – (N_atoms, ndim) shape.
- **cells** (*tf.Tensor*) – (n_cells, ndim) shape.

Returns
closest_inds – Of shape (n_cells, M_nbrs)

Return type
tf.Tensor

`get_cells_for_atoms`(*coords*, *cells*)

Compute the cells each atom belongs to.

Parameters

- **`coords`** (*`tf.Tensor`*) – Shape (N_atoms, ndim)
- **`cells`** (*`tf.Tensor`*) – (n_cells, ndim) shape.

Returns
cells_for_atoms – Shape (N_atoms, 1)

Return type
tf.Tensor

`get_neighbor_cells`(*cells*)

Compute neighbors of cells in grid.

# TODO(rbharath): Do we need to handle periodic boundary conditions properly here? # TODO(rbharath): This doesn't handle boundaries well. We hard-code # looking for n_nbr_cells neighbors, which isn't right for boundary cells in # the cube.

Parameters
`cells` (*`tf.Tensor`*) – (n_cells, ndim) shape.

Returns
nbr_cells – (n_cells, n_nbr_cells)

Return type
tf.Tensor

`get_cells`()

Returns the locations of all grid points in box.

Suppose start is -10 Angstrom, stop is 10 Angstrom, nbr_cutoff is 1. Then would return a list of length 20^3 whose entries would be [(-10, -10, -10), (-10, -10, -9), . . . , (9, 9, 9)]

Returns
cells – (n_cells, ndim) shape.

Return type
tf.Tensor

`class AtomicConvolution`(**args*, ***kwargs*)

Implements the atomic convolutional transform introduced in

Gomes, Joseph, et al. "Atomic convolutional networks for predicting protein-ligand binding affinity." arXiv preprint arXiv:1703.10603 (2017).

At a high level, this transform performs a graph convolution on the nearest neighbors graph in 3D space.

`__init__`(*atom_types=None*, *radial_params=[]*, *boxsize=None*, ***kwargs*)

Atomic convolution layer

N = max_num_atoms, M = max_num_neighbors, B = batch_size, d = num_features l = num_radial_filters * num_atom_types

Parameters

- **`atom_types`** (*`list or None`*) – Of length a, where a is number of atom types for filtering.

- **radial_params** (*list*) – Of length l, where l is number of radial filters learned.
- **boxsize** (*float or None*) – Simulation box length [Angstrom].

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*)

Parameters

- **X** (*tf.Tensor of shape (B, N, d)*) – Coordinates/features.
- **Nbrs** (*tf.Tensor of shape (B, N, M)*) – Neighbor list.
- **Nbrs_Z** (*tf.Tensor of shape (B, N, M)*) – Atomic numbers of neighbor atoms.

Returns
layer – A new tensor representing the output of the atomic conv layer

Return type
tf.Tensor of shape (B, N, l)

radial_symmetry_function(*R*, *rc*, *rs*, *e*)

Calculates radial symmetry function.

B = batch_size, N = max_num_atoms, M = max_num_neighbors, d = num_filters

Parameters

- **R** (*tf.Tensor of shape (B, N, M)*) – Distance matrix.
- **rc** (*float*) – Interaction cutoff [Angstrom].
- **rs** (*float*) – Gaussian distance matrix mean.
- **e** (*float*) – Gaussian distance matrix width.

Returns
retval – Radial symmetry function (before summation)

Return type
tf.Tensor of shape (B, N, M)

radial_cutoff(*R*, *rc*)

Calculates radial cutoff matrix.

B = batch_size, N = max_num_atoms, M = max_num_neighbors

Parameters

- **[B** (*R*) – Distance matrix.
- **N** (*tf.Tensor*) – Distance matrix.
- **M]** (*tf.Tensor*) – Distance matrix.
- **rc** (*tf.Variable*) – Interaction cutoff [Angstrom].

Returns
FC [B, N, M] – Radial cutoff matrix.

Return type
tf.Tensor

gaussian_distance_matrix(*R*, *rs*, *e*)

Calculates gaussian distance matrix.

B = batch_size, N = max_num_atoms, M = max_num_neighbors

Parameters

- **[B** (*R*) – Distance matrix.
- **N** (*tf.Tensor*) – Distance matrix.
- **M]** (*tf.Tensor*) – Distance matrix.
- **rs** (*tf.Variable*) – Gaussian distance matrix mean.
- **e** (*tf.Variable*) – Gaussian distance matrix width (e = .5/std**2).

Returns
retval [B, N, M] – Gaussian distance matrix.

Return type
tf.Tensor

distance_tensor(*X*, *Nbrs*, *boxsize*, *B*, *N*, *M*, *d*)

Calculates distance tensor for batch of molecules.

B = batch_size, N = max_num_atoms, M = max_num_neighbors, d = num_features

Parameters

- **X** (*tf.Tensor of shape (B, N, d)*) – Coordinates/features tensor.
- **Nbrs** (*tf.Tensor of shape (B, N, M)*) – Neighbor list tensor.
- **boxsize** (*float or None*) – Simulation box length [Angstrom].

Returns
D – Coordinates/features distance tensor.

Return type
tf.Tensor of shape (B, N, M, d)

`distance_matrix`(*D*)

Calcuates the distance matrix from the distance tensor

B = batch_size, N = max_num_atoms, M = max_num_neighbors, d = num_features

Parameters
`D` (*tf.Tensor of shape (B, N, M, d)*) – Distance tensor.

Returns
R – Distance matrix.

Return type
tf.Tensor of shape (B, N, M)

class `AlphaShareLayer`(**args*, ***kwargs*)

Part of a sluice network. Adds alpha parameters to control sharing between the main and auxillary tasks

Factory method AlphaShare should be used for construction

Parameters
`in_layers` (*list of Layers or tensors*) – tensors in list must be the same size and list must include two or more tensors

Returns

- **out_tensor** (*a tensor with shape [len(in_layers), x, y] where x, y were the original layer dimensions*)
- *Distance matrix.*

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- `*args` – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- `**kwargs` – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns

A tensor or list/tuple of tensors.

`class SluiceLoss(*args, **kwargs)`

Calculates the loss in a Sluice Network Every input into an AlphaShare should be used in SluiceLoss

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **`inputs`** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.

  – The SavedModel input specification is generated using *inputs* only.

  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **`*args`** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **`**kwargs`** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class BetaShare`(**args*, ***kwargs*)

Part of a sluice network. Adds beta params to control which layer outputs are used for prediction

Parameters
`in_layers` (*list of Layers or tensors*) – tensors in list must be the same size and list must include two or more tensors

Returns
output_layers – Distance matrix.

Return type
list of Layers or tensors with same size as in_layers

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

Size of input layers must all be the same

`class ANIFeat`(**args*, ***kwargs*)

Performs transform from 3D coordinates to ANI symmetry functions

`__init__`(*max_atoms=23*, *radial_cutoff=4.6*, *angular_cutoff=3.1*, *radial_length=32*, *angular_length=8*, *atom_cases=[1, 6, 7, 8, 16]*, *atomic_number_differentiated=True*, *coordinates_in_bohr=True*, ***kwargs*)

Only X can be transformed

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`call`(*inputs*)

In layers should be of shape dtype tf.float32, (None, self.max_atoms, 4)

`distance_matrix`(*coordinates*, *flags*)

Generate distance matrix

`distance_cutoff`(*d*, *cutoff*, *flags*)

Generate distance matrix with trainable cutoff

`radial_symmetry`(*d_cutoff*, *d*, *atom_numbers*)

Radial Symmetry Function

`angular_symmetry`(*d_cutoff*, *d*, *atom_numbers*, *coordinates*)

Angular Symmetry Function

`class GraphEmbedPoolLayer`(**args*, ***kwargs*)

GraphCNNPool Layer from Robust Spatial Filtering with Graph Convolutional Neural Networks https://arxiv.org/abs/1703.00792

This is a learnable pool operation It constructs a new adjacency matrix for a graph of specified number of nodes.

This differs from our other pool operations which set vertices to a function value without altering the adjacency matrix.

..math:: V_{emb} = SpatialGraphCNN({V_{in}}) ..math:: V_{out} = sigma(V_{emb})^{T} * V_{in} ..math:: A_{out} = V_{emb}^{T} * A_{in} * V_{emb}

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

Parameters

- **num_filters** (*int*) – Number of filters to have in the output
- **in_layers** (*list of Layers or tensors*) – [V, A, mask] V are the vertex features must be of shape (batch, vertex, channel)
- **graph** (*A are the adjacency matrixes for each*) – Shape (batch, from_vertex, adj_matrix, to_vertex)
- **optional** (*mask is*) –
- **the** (*to be used when not every graph has*) –
- **vertices** (*same number of*) –

Returns

- Returns a *tf.tensor* with a graph convolution applied
- The shape will be *(batch, vertex, self.num_filters)*.

class GraphCNN(**args*, ***kwargs*)

GraphCNN Layer from Robust Spatial Filtering with Graph Convolutional Neural Networks https://arxiv.org/abs/1703.00792

Spatial-domain convolutions can be defined as H = h_0I + h_1A + h_2A^2 + … + hkAk, H R**(N×N)

We approximate it by H h_0I + h_1A

We can define a convolution as applying multiple these linear filters over edges of different types (think up, down, left, right, diagonal in images) Where each edge type has its own adjacency matrix H h_0I + h_1A_1 + h_2A_2 + … h_(L1)A_(L1)

V_out = sum_{c=1}^{C} H^{c} V^{c} + b

__init__(*num_filters*, ***kwargs*)

Parameters

- **num_filters** (*int*) – Number of filters to have in the output
- **in_layers** (*list of Layers or tensors*) – [V, A, mask] V are the vertex features must be of shape (batch, vertex, channel)
- **graph** (*A are the adjacency matrixes for each*) – Shape (batch, from_vertex, adj_matrix, to_vertex)
- **optional** (*mask is*) –
- **the** (*to be used when not every graph has*) –
- **vertices** (*same number of*) –
- **Returns** (*tf.tensor*) –
- **applied** (*Returns a tf.tensor with a graph convolution*) –
- **(batch** (*The shape will be*) –
- **vertex** –
- **self.num_filters)** –

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **`inputs`** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.

  – The SavedModel input specification is generated using *inputs* only.

  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- **`*args`** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- **`**kwargs`** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class Highway`(**args*, ***kwargs*)

Create a highway layer. y = H(x) * T(x) + x * (1 - T(x))

H(x) = activation_fn(matmul(W_H, x) + b_H) is the non-linear transformed output T(x) = sigmoid(matmul(W_T, x) + b_T) is the transform gate

Implementation based on paper

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber. "Highway networks." arXiv preprint arXiv:1505.00387 (2015).

This layer expects its input to be a two dimensional tensor of shape (batch size, # input features). Outputs will be in the same shape.

`__init__`(*activation_fn='relu'*, *biases_initializer='zeros'*, *weights_initializer=None*, ***kwargs*)

Parameters

- **activation_fn** (`object`) – the Tensorflow activation function to apply to the output
- **biases_initializer** (`callable object`) – the initializer for bias values. This may be None, in which case the layer will not include biases.
- **weights_initializer** (`callable object`) – the initializer for weight values

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.

  – The SavedModel input specification is generated using *inputs* only.

  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- `*args` – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- `**kwargs` – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

class WeaveLayer(**args*, ***kwargs*)

This class implements the core Weave convolution from the Google graph convolution paper [1]_

This model contains atom features and bond features separately.Here, bond features are also called pair features. There are 2 types of transformation, atom->atom, atom->pair, pair->atom, pair->pair that this model implements.

Examples

This layer expects 4 inputs in a list of the form *[atom_features, pair_features, pair_split, atom_to_pair]*. We'll walk through the structure of these inputs. Let's start with some basic definitions.

```
>>> import deepchem as dc
>>> import numpy as np
```

Suppose you have a batch of molecules

```
>>> smiles = ["CCC", "C"]
```

Note that there are 4 atoms in total in this system. This layer expects its input molecules to be batched together.

```
>>> total_n_atoms = 4
```

Let's suppose that we have a featurizer that computes *n_atom_feat* features per atom.

```
>>> n_atom_feat = 75
```

Then conceptually, *atom_feat* is the array of shape *(total_n_atoms, n_atom_feat)* of atomic features. For simplicity, let's just go with a random such matrix.

```
>>> atom_feat = np.random.rand(total_n_atoms, n_atom_feat)
```

Let's suppose we have *n_pair_feat* pairwise features

```
>>> n_pair_feat = 14
```

For each molecule, we compute a matrix of shape *(n_atoms*n_atoms, n_pair_feat)* of pairwise features for each pair of atoms in the molecule. Let's construct this conceptually for our example.

```
>>> pair_feat = [np.random.rand(3*3, n_pair_feat), np.random.rand(1*1, n_pair_feat)]
>>> pair_feat = np.concatenate(pair_feat, axis=0)
>>> pair_feat.shape
(10, 14)
```

pair_split is an index into *pair_feat* which tells us which atom each row belongs to. In our case, we hve

```
>>> pair_split = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3])
```

That is, the first 9 entries belong to "CCC" and the last entry to "C". The final entry *atom_to_pair* goes in a little more in-depth than *pair_split* and tells us the precise pair each pair feature belongs to. In our case

```
>>> atom_to_pair = np.array([[0, 0],
...                          [0, 1],
...                          [0, 2],
...                          [1, 0],
...                          [1, 1],
```

(continues on next page)

(continued from previous page)

```
...                          [1, 2],
...                          [2, 0],
...                          [2, 1],
...                          [2, 2],
...                          [3, 3]])
```

Let's now define the actual layer

```
>>> layer = WeaveLayer()
```

And invoke it

```
>>> [A, P] = layer([atom_feat, pair_feat, pair_split, atom_to_pair])
```

The weave layer produces new atom/pair features. Let's check their shapes

```
>>> A = np.array(A)
>>> A.shape
(4, 50)
>>> P = np.array(P)
>>> P.shape
(10, 50)
```

The 4 is *total_num_atoms* and the 10 is the total number of pairs. Where does *50* come from? It's from the default arguments *n_atom_input_feat* and *n_pair_input_feat*.

References

__init__(*n_atom_input_feat: int = 75*, *n_pair_input_feat: int = 14*, *n_atom_output_feat: int = 50*, *n_pair_output_feat: int = 50*, *n_hidden_AA: int = 50*, *n_hidden_PA: int = 50*, *n_hidden_AP: int = 50*, *n_hidden_PP: int = 50*, *update_pair: bool = True*, *init: str = 'glorot_uniform'*, *activation: str = 'relu'*, *batch_normalize: bool = True*, *batch_normalize_kwargs: Dict = {'renorm': True}*, ***kwargs*)

Parameters

- **n_atom_input_feat** (*int, optional (default 75)*) – Number of features for each atom in input.
- **n_pair_input_feat** (*int, optional (default 14)*) – Number of features for each pair of atoms in input.
- **n_atom_output_feat** (*int, optional (default 50)*) – Number of features for each atom in output.
- **n_pair_output_feat** (*int, optional (default 50)*) – Number of features for each pair of atoms in output.
- **n_hidden_AA** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_hidden_PA** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_hidden_AP** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer

- **n_hidden_PP** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **update_pair** (*bool, optional (default True)*) – Whether to calculate for pair features, could be turned off for last layer
- **init** (*str, optional (default 'glorot_uniform')*) – Weight initialization for filters.
- **activation** (*str, optional (default 'relu')*) – Activation function applied
- **batch_normalize** (*bool, optional (default True)*) – If this is turned on, apply batch normalization before applying activation functions on convolutional layers.
- **batch_normalize_kwargs** (Dict, optional (default *{renorm=True}*)) – Batch normalization is a complex layer which has many potential argumentswhich change behavior. This layer accepts user-defined parameters which are passed to all *BatchNormalization* layers in *WeaveModel*, *WeaveLayer*, and *WeaveGather*.

get_config() → Dict

Returns config dictionary for this layer.

build(*input_shape*)

Construct internal trainable weights.

Parameters

input_shape (*tuple*) – Ignored since we don't need the input shape to create internal weights.

call(*inputs: List*) → List

Creates weave tensors.

Parameters

inputs (*List*) – Should contain 4 tensors [atom_features, pair_features, pair_split, atom_to_pair]

class WeaveGather(**args*, ***kwargs*)

Implements the weave-gathering section of weave convolutions.

Implements the gathering layer from [1]_. The weave gathering layer gathers per-atom features to create a molecule-level fingerprint in a weave convolutional network. This layer can also performs Gaussian histogram expansion as detailed in [1]_. Note that the gathering function here is simply addition as in [1]_>

Examples

This layer expects 2 inputs in a list of the form *[atom_features, pair_features]*. We'll walk through the structure of these inputs. Let's start with some basic definitions.

```
>>> import deepchem as dc
>>> import numpy as np
```

Suppose you have a batch of molecules

```
>>> smiles = ["CCC", "C"]
```

Note that there are 4 atoms in total in this system. This layer expects its input molecules to be batched together.

```
>>> total_n_atoms = 4
```

Let's suppose that we have *n_atom_feat* features per atom.

```
>>> n_atom_feat = 75
```

Then conceptually, *atom_feat* is the array of shape *(total_n_atoms, n_atom_feat)* of atomic features. For simplicity, let's just go with a random such matrix.

```
>>> atom_feat = np.random.rand(total_n_atoms, n_atom_feat)
```

We then need to provide a mapping of indices to the atoms they belong to. In ours case this would be

```
>>> atom_split = np.array([0, 0, 0, 1])
```

Let's now define the actual layer

```
>>> gather = WeaveGather(batch_size=2, n_input=n_atom_feat)
>>> output_molecules = gather([atom_feat, atom_split])
>>> len(output_molecules)
2
```

References

Note: This class requires *tensorflow_probability* to be installed.

`__init__`(*batch_size: int*, *n_input: int = 128*, *gaussian_expand: bool = True*, *compress_post_gaussian_expansion: bool = False*, *init: str = 'glorot_uniform'*, *activation: str = 'tanh'*, ***kwargs*)

Parameters

- **`batch_size`** (*`int`*) – number of molecules in a batch
- **`n_input`** (*`int, optional (default 128)`*) – number of features for each input molecule
- **`gaussian_expand`** (*`boolean, optional (default True)`*) – Whether to expand each dimension of atomic features by gaussian histogram
- **`compress_post_gaussian_expansion`** (*`bool, optional (default False)`*) – If True, compress the results of the Gaussian expansion back to the original dimensions of the input by using a linear layer with specified activation function. Note that this compression was not in the original paper, but was present in the original DeepChem implementation so is left present for backwards compatibility.
- **`init`** (*`str, optional (default 'glorot_uniform')`*) – Weight initialization for filters if *compress_post_gaussian_expansion* is True.
- **`activation`** (*`str, optional (default 'tanh')`*) – Activation function applied for filters if *compress_post_gaussian_expansion* is True. Should be recognizable by *tf.keras.activations*.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs: List*) → List

Creates weave tensors.

Parameters
`inputs` (`List`) – Should contain 2 tensors [atom_features, atom_split]

Returns
output_molecules – Each entry in this list is of shape *(self.n_inputs,)*

Return type
List

`gaussian_histogram`(*x*)

Expands input into a set of gaussian histogram bins.

Parameters
`x` (`tf.Tensor`) – Of shape *(N, n_feat)*

Examples

This method uses 11 bins spanning portions of a Gaussian with zero mean and unit standard deviation.

```
>>> gaussian_memberships = [(-1.645, 0.283), (-1.080, 0.170),
...                         (-0.739, 0.134), (-0.468, 0.118),
...                         (-0.228, 0.114), (0., 0.114),
...                         (0.228, 0.114), (0.468, 0.118),
...                         (0.739, 0.134), (1.080, 0.170),
...                         (1.645, 0.283)]
```

We construct a Gaussian at *gaussian_memberships[i][0]* with standard deviation *gaussian_memberships[i][1]*. Each feature in *x* is assigned the probability of falling in each Gaussian, and probabilities are normalized across the 11 different Gaussians.

Returns
outputs – Of shape *(N, 11*n_feat)*

Return type
tf.Tensor

`class DTNNEmbedding(*args, **kwargs)`

`__init__(n_embedding=30, periodic_table_length=30, init='glorot_uniform', **kwargs)`

Parameters

- **n_embedding** (`int, optional`) – Number of features for each atom
- **periodic_table_length** (`int, optional`) – Length of embedding, 83=Bi
- **init** (`str, optional`) – Weight initialization for filters.

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build(input_shape)`

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call(inputs)`

parent layers: atom_number

`class DTNNStep(*args, **kwargs)`

`__init__(n_embedding=30, n_distance=100, n_hidden=60, init='glorot_uniform', activation='tanh', **kwargs)`

Parameters

- **n_embedding** (`int, optional`) – Number of features for each atom
- **n_distance** (`int, optional`) – granularity of distance matrix
- **n_hidden** (`int, optional`) – Number of nodes in hidden layer
- **init** (`str, optional`) – Weight initialization for filters.
- **activation** (`str, optional`) – Activation function applied

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build(`*input_shape*`)`

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call(`*inputs*`)`

parent layers: atom_features, distance, distance_membership_i, distance_membership_j

`class DTNNGather(`**args*, ***kwargs*`)`

`__init__(`*n_embedding=30*, *n_outputs=100*, *layer_sizes=[100]*, *output_activation=True*, *init='glorot_uniform'*, *activation='tanh'*, ***kwargs*`)`

Parameters

- **n_embedding** (*int, optional*) – Number of features for each atom
- **n_outputs** (*int, optional*) – Number of features for each molecule(output)
- **layer_sizes** (*list of int, optional(default=[1000])*) – Structure of hidden layer(s)
- **init** (*str, optional*) – Weight initialization for filters.
- **activation** (*str, optional*) – Activation function applied

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters

input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*)

parent layers: atom_features, atom_membership

class **DAGLayer**(**args*, ***kwargs*)

DAG computation layer.

This layer generates a directed acyclic graph for each atom in a molecule. This layer is based on the algorithm from the following paper:

Lusci, Alessandro, Gianluca Pollastri, and Pierre Baldi. "Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules." Journal of chemical information and modeling 53.7 (2013): 1563-1575.

This layer performs a sort of inward sweep. Recall that for each atom, a DAG is generated that "points inward" to that atom from the undirected molecule graph. Picture this as "picking up" the atom as the vertex and using the natural tree structure that forms from gravity. The layer "sweeps inwards" from the leaf nodes of the DAG upwards to the atom. This is batched so the transformation is done for each atom.

__init__(*n_graph_feat=30*, *n_atom_feat=75*, *max_atoms=50*, *layer_sizes=[100]*, *init='glorot_uniform'*, *activation='relu'*, *dropout=None*, *batch_size=64*, ***kwargs*)

Parameters

- **n_graph_feat** (`int, optional`) – Number of features for each node(and the whole grah).
- **n_atom_feat** (`int, optional`) – Number of features listed per atom.
- **max_atoms** (`int, optional`) – Maximum number of atoms in molecules.
- **layer_sizes** (`list of int, optional(default=[100])`) – List of hidden layer size(s): length of this list represents the number of hidden layers, and each element is the width of corresponding hidden layer.
- **init** (`str, optional`) – Weight initialization for filters.
- **activation** (`str, optional`) – Activation function applied.
- **dropout** (`float, optional`) – Dropout probability in hidden layer(s).
- **batch_size** (`int, optional`) – number of molecules in a batch.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

"Construct internal trainable weights.

`call`(*inputs*, *training=True*)

parent layers: atom_features, parents, calculation_orders, calculation_masks, n_atoms

class `DAGGather`(**args*, ***kwargs*)

`__init__`(*n_graph_feat=30*, *n_outputs=30*, *max_atoms=50*, *layer_sizes=[100]*, *init='glorot_uniform'*, *activation='relu'*, *dropout=None*, ***kwargs*)

DAG vector gathering layer

Parameters

- **n_graph_feat** (`int, optional`) – Number of features for each atom.
- **n_outputs** (`int, optional`) – Number of features for each molecule.
- **max_atoms** (`int, optional`) – Maximum number of atoms in molecules.
- **layer_sizes** (`list of int, optional`) – List of hidden layer size(s): length of this list represents the number of hidden layers, and each element is the width of corresponding hidden layer.
- **init** (`str, optional`) – Weight initialization for filters.
- **activation** (`str, optional`) – Activation function applied.
- **dropout** (`float, optional`) – Dropout probability in the hidden layer(s).

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*, *training=True*)

parent layers: atom_features, membership

`class MessagePassing`(**args*, ***kwargs*)

General class for MPNN default structures built according to https://arxiv.org/abs/1511.06391

`__init__`(*T*, *message_fn='enn'*, *update_fn='gru'*, *n_hidden=100*, ***kwargs*)

Parameters

- **T** (*int*) – Number of message passing steps
- **message_fn** (*str, optional*) – message function in the model
- **update_fn** (*str, optional*) – update function in the model
- **n_hidden** (*int, optional*) – number of hidden units in the passing phase

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build`(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call`(*inputs*)

Perform T steps of message passing

`class EdgeNetwork`(**args*, ***kwargs*)

Submodule for Message Passing

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build(input_shape)`

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call(inputs)`

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  – NumPy array or Python scalar values in *inputs* get cast as tensors.

  – Keras mask metadata is only collected from *inputs*.

  – Layers are built (*build(input_shape)* method) using shape info from *inputs* only.

  – *input_spec* compatibility is only checked against *inputs*.

  – Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.

  – The SavedModel input specification is generated using *inputs* only.

  – Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- `*args` – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.

- `**kwargs` – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  – *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class GatedRecurrentUnit(*args, **kwargs)`

Submodule for Message Passing

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

`build(input_shape)`

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
`input_shape` – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

`call(inputs)`

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- `inputs` – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- `*args` – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- `**kwargs` – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

`class SetGather(*args, **kwargs)`

set2set gather layer for graph-based model

Models using this layer must set *pad_batches=True*.

`__init__(M, batch_size, n_hidden=100, init='orthogonal', **kwargs)`

Parameters

- `M` (*int*) – Number of LSTM steps
- `batch_size` (*int*) – Number of samples in a batch(all batches must have same size)
- `n_hidden` (*int, optional*) – number of hidden units in the passing phase

`get_config()`

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Note that *get_config()* does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Returns
Python dictionary.

build(*input_shape*)

Creates the variables of the layer (for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call. It is invoked automatically before the first execution of *call()*.

This is typically used to create the weights of *Layer* subclasses (at the discretion of the subclass implementer).

Parameters
input_shape – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*)

Perform M steps of set2set gather,

Detailed descriptions in: https://arxiv.org/abs/1511.06391

3.23.3 Torch Layers

class AtomicConv(*n_tasks: int*, *frag1_num_atoms: int = 70*, *frag2_num_atoms: int = 634*, *complex_num_atoms: int = 701*, *max_num_neighbors: int = 12*, *batch_size: int = 24*, *atom_types: Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0]*, *radial: Sequence[Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]]*, *layer_sizes=[100]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = ['relu']*, *init: str = 'trunc_normal_'*, ***kwargs*)

Implements an Atomic Convolution Model.

The atomic convolutional networks function as a variant of graph convolutions. The difference is that the "graph" here is the nearest neighbors graph in 3D space [1]. The AtomicConvModule leverages these connections in 3D space to train models that learn to predict energetic states starting from the spatial geometry of the model.

References

Examples

```
>>> n_tasks = 1
>>> frag1_num_atoms = 70
>>> frag2_num_atoms = 634
>>> complex_num_atoms = 701
>>> max_num_neighbors = 12
>>> batch_size = 24
>>> atom_types = [
...     6, 7., 8., 9., 11., 12., 15., 16., 17., 20., 25., 30., 35., 53.,
...     -1.
... ]
>>> radial = [[
...     1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5,
```

(continues on next page)

(continued from previous page)

```
...     8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0
... ], [0.0, 4.0, 8.0], [0.4]]
>>> layer_sizes = [32, 32, 16]
>>> acnn_model = AtomicConv(n_tasks=n_tasks,
... frag1_num_atoms=frag1_num_atoms,
... frag2_num_atoms=frag2_num_atoms,
... complex_num_atoms=complex_num_atoms,
... max_num_neighbors=max_num_neighbors,
... batch_size=batch_size,
... atom_types=atom_types,
... radial=radial,
... layer_sizes=layer_sizes)
```

`__init__`(*n_tasks: int*, *frag1_num_atoms: int = 70*, *frag2_num_atoms: int = 634*, *complex_num_atoms: int = 701*, *max_num_neighbors: int = 12*, *batch_size: int = 24*, *atom_types: Sequence[float] = [6, 7.0, 8.0, 9.0, 11.0, 12.0, 15.0, 16.0, 17.0, 20.0, 25.0, 30.0, 35.0, 53.0, -1.0]*, *radial: Sequence[Sequence[float]] = [[1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0], [0.0, 4.0, 8.0], [0.4]]*, *layer_sizes=[100]*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = ['relu']*, *init: str = 'trunc_normal_'*, ***kwargs*) → None

Parameters

- **`n_tasks`** (`int`) – number of tasks
- **`frag1_num_atoms`** (`int`) – Number of atoms in first fragment
- **`frag2_num_atoms`** (`int`) – Number of atoms in sec
- **`max_num_neighbors`** (`int`) – Maximum number of neighbors possible for an atom. Recall neighbors are spatial neighbors.
- **`atom_types`** (`list`) – List of atoms recognized by model. Atoms are indicated by their nuclear numbers.
- **`radial`** (`list`) – Radial parameters used in the atomic convolution transformation.
- **`layer_sizes`** (`list`) – the size of each dense layer in the network. The length of this list determines the number of layers.
- **`weight_init_stddevs`** (`list or float`) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`bias_init_consts`** (`list or float`) – the value to initialize the biases in each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`dropouts`** (`list or float`) – the dropout probability to use for each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.
- **`activation_fns`** (`list or object`) – the Tensorflow activation function to apply to each layer. The length of this list should equal len(layer_sizes). Alternatively, this may be a single value instead of a list, where the same value is used for every layer.

forward(*inputs: Tensor | Sequence[Tensor]*)

Parameters
inputs (*torch.Tensor*) – Input Tensor

Returns
Output for each label.

Return type
torch.Tensor

class MultilayerPerceptron(*d_input: int*, *d_output: int*, *d_hidden: tuple | None = None*, *dropout: float = 0.0*, *batch_norm: bool = False*, *batch_norm_momentum: float = 0.1*, *activation_fn: Callable | str = 'relu'*, *skip_connection: bool = False*, *weighted_skip: bool = True*)

A simple fully connected feed-forward network, otherwise known as a multilayer perceptron (MLP).

Examples

```
>>> model = MultilayerPerceptron(d_input=10, d_hidden=(2,3), d_output=2, dropout=0.
↪0, activation_fn='relu')
>>> x = torch.ones(2, 10)
>>> out = model(x)
>>> print(out.shape)
torch.Size([2, 2])
```

__init__(*d_input: int*, *d_output: int*, *d_hidden: tuple | None = None*, *dropout: float = 0.0*, *batch_norm: bool = False*, *batch_norm_momentum: float = 0.1*, *activation_fn: Callable | str = 'relu'*, *skip_connection: bool = False*, *weighted_skip: bool = True*)

Initialize the model.

Parameters

- **d_input** (*int*) – the dimension of the input layer
- **d_output** (*int*) – the dimension of the output layer
- **d_hidden** (*tuple*) – the dimensions of the hidden layers
- **dropout** (*float*) – the dropout probability
- **batch_norm** (*bool*) – whether to use batch normalization
- **batch_norm_momentum** (*float*) – the momentum for batch normalization
- **activation_fn** (*str*) – the activation function to use in the hidden layers
- **skip_connection** (*bool*) – whether to add a skip connection from the input to the output
- **weighted_skip** (*bool*) – whether to add a weighted skip connection from the input to the output

build_layers()

Build the layers of the model, iterating through the hidden dimensions to produce a list of layers.

forward(*x: Tensor*) → Tensor

Forward pass of the model.

class CNNModule(*n_tasks: int*, *n_features: int*, *dims: int*, *layer_filters: List[int] = [100]*, *kernel_size: int | Sequence[int] = 5*, *strides: int | Sequence[int] = 1*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *pool_type: str = 'max'*, *mode: str = 'classification'*, *n_classes: int = 2*, *uncertainty: bool = False*, *residual: bool = False*, *padding: int | str = 'valid'*)

A 1, 2, or 3 dimensional convolutional network for either regression or classification. The network consists of the following sequence of layers: - A configurable number of convolutional layers - A global pooling layer (either max pool or average pool) - A final fully connected layer to compute the output It optionally can compose the model from pre-activation residual blocks, as described in https://arxiv.org/abs/1603.05027, rather than a simple stack of convolution layers. This often leads to easier training, especially when using a large number of layers. Note that residual blocks can only be used when successive layers have the same output shape. Wherever the output shape changes, a simple convolution layer will be used even if residual=True. .. rubric:: Examples

```
>>> model = CNNModule(n_tasks=5, n_features=8, dims=2, layer_filters=[3,8,8,16],␣
↪kernel_size=3, n_classes = 7, mode='classification', uncertainty=False, padding=
↪'same')
>>> x = torch.ones(2, 224, 224, 8)
>>> x = model(x)
>>> for tensor in x:
...     print(tensor.shape)
torch.Size([2, 5, 7])
torch.Size([2, 5, 7])
```

__init__(*n_tasks: int*, *n_features: int*, *dims: int*, *layer_filters: List[int] = [100]*, *kernel_size: int | Sequence[int] = 5*, *strides: int | Sequence[int] = 1*, *weight_init_stddevs: float | Sequence[float] = 0.02*, *bias_init_consts: float | Sequence[float] = 1.0*, *dropouts: float | Sequence[float] = 0.5*, *activation_fns: Callable | str | Sequence[Callable | str] = 'relu'*, *pool_type: str = 'max'*, *mode: str = 'classification'*, *n_classes: int = 2*, *uncertainty: bool = False*, *residual: bool = False*, *padding: int | str = 'valid'*) → None

Create a CNN.

Parameters

- **n_tasks** (*int*) – number of tasks
- **n_features** (*int*) – number of features
- **dims** (*int*) – the number of dimensions to apply convolutions over (1, 2, or 3)
- **layer_filters** (*list*) – the number of output filters for each convolutional layer in the network. The length of this list determines the number of layers.
- **kernel_size** (*int, tuple, or list*) – a list giving the shape of the convolutional kernel for each layer. Each element may be either an int (use the same kernel width for every dimension) or a tuple (the kernel width along each dimension). Alternatively this may be a single int or tuple instead of a list, in which case the same kernel shape is used for every layer.
- **strides** (*int, tuple, or list*) – a list giving the stride between applications of the kernel for each layer. Each element may be either an int (use the same stride for every dimension) or a tuple (the stride along each dimension). Alternatively this may be a single int or tuple instead of a list, in which case the same stride is used for every layer.
- **weight_init_stddevs** (*list or float*) – the standard deviation of the distribution to use for weight initialization of each layer. The length of this list should equal len(layer_filters)+1, where the final element corresponds to the dense layer. Alternatively

this may be a single value instead of a list, in which case the same value is used for every layer.

- **bias_init_consts** (*list or float*) – the value to initialize the biases in each layer to. The length of this list should equal len(layer_filters)+1, where the final element corresponds to the dense layer. Alternatively this may be a single value instead of a list, in which case the same value is used for every layer.
- **dropouts** (*list or float*) – the dropout probability to use for each layer. The length of this list should equal len(layer_filters). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer
- **activation_fns** (*str or list*) – the torch activation function to apply to each layer. The length of this list should equal len(layer_filters). Alternatively this may be a single value instead of a list, in which case the same value is used for every layer, 'relu' by default
- **pool_type** (*str*) – the type of pooling layer to use, either 'max' or 'average'
- **mode** (*str*) – Either 'classification' or 'regression'
- **n_classes** (*int*) – the number of classes to predict (only used in classification mode)
- **uncertainty** (*bool*) – if True, include extra outputs and loss terms to enable the uncertainty in outputs to be predicted
- **residual** (*bool*) – if True, the model will be composed of pre-activation residual blocks instead of a simple stack of convolutional layers.
- **padding** (*str, int or tuple*) – the padding to use for convolutional layers, either 'valid' or 'same'

forward(*inputs: Tensor | Sequence[Tensor]*) → List[Any]

Parameters
x (*torch.Tensor*) – Input Tensor

Returns
Output as per use case : regression/classification

Return type
torch.Tensor

class **ScaleNorm**(*scale: float*, *eps: float = 1e-05*)

Apply Scale Normalization to input.

The ScaleNorm layer first computes the square root of the scale, then computes the matrix/vector norm of the input tensor. The norm value is calculated as *sqrt(scale) / matrix norm*. Finally, the result is returned as *input_tensor * norm value*.

This layer can be used instead of LayerNorm when a scaled version of the norm is required. Instead of performing the scaling operation (*scale / norm*) in a lambda-like layer, we are defining it within this layer to make prototyping more efficient.

References

Examples

```
>>> from deepchem.models.torch_models.layers import ScaleNorm
>>> scale = 0.35
>>> layer = ScaleNorm(scale)
>>> input_tensor = torch.tensor([[1.269, 39.36], [0.00918, -9.12]])
>>> output_tensor = layer(input_tensor)
```

`__init__`(*scale: float*, *eps: float = 1e-05*)

Initialize a ScaleNorm layer.

Parameters

- **scale** (`float`) – Scale magnitude.
- **eps** (`float`) – Epsilon value. Default = 1e-5.

`forward`(*x: Tensor*) → Tensor

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class `MATEncoderLayer`(*dist_kernel: str = 'softmax'*, *lambda_attention: float = 0.33*, *lambda_distance: float = 0.33*, *h: int = 16*, *sa_hsize: int = 1024*, *sa_dropout_p: float = 0.0*, *output_bias: bool = True*, *d_input: int = 1024*, *d_hidden: int = 1024*, *d_output: int = 1024*, *activation: str = 'leakyrelu'*, *n_layers: int = 1*, *ff_dropout_p: float = 0.0*, *encoder_hsize: int = 1024*, *encoder_dropout_p: float = 0.0*)

Encoder layer for use in the Molecular Attention Transformer [1]_.

The MATEncoder layer primarily consists of a self-attention layer (MultiHeadedMATAttention) and a feed-forward layer (PositionwiseFeedForward). This layer can be stacked multiple times to form an encoder.

References

Examples

```
>>> from rdkit import Chem
>>> import torch
>>> import deepchem
>>> from deepchem.models.torch_models.layers import MATEmbedding, MATEncoderLayer
>>> input_smile = "CC"
>>> feat = deepchem.feat.MATFeaturizer()
>>> out = feat.featurize(input_smile)
>>> node = torch.tensor(out[0].node_features).float().unsqueeze(0)
>>> adj = torch.tensor(out[0].adjacency_matrix).float().unsqueeze(0)
>>> dist = torch.tensor(out[0].distance_matrix).float().unsqueeze(0)
>>> mask = torch.sum(torch.abs(node), dim=-1) != 0
```

(continues on next page)

(continued from previous page)

```
>>> layer = MATEncoderLayer()
>>> op = MATEmbedding()(node)
>>> output = layer(op, mask, adj, dist)
```

__init__(*dist_kernel: str = 'softmax'*, *lambda_attention: float = 0.33*, *lambda_distance: float = 0.33*, *h: int = 16*, *sa_hsize: int = 1024*, *sa_dropout_p: float = 0.0*, *output_bias: bool = True*, *d_input: int = 1024*, *d_hidden: int = 1024*, *d_output: int = 1024*, *activation: str = 'leakyrelu'*, *n_layers: int = 1*, *ff_dropout_p: float = 0.0*, *encoder_hsize: int = 1024*, *encoder_dropout_p: float = 0.0*)

Initialize a MATEncoder layer.

Parameters

- **dist_kernel** (*str*) – Kernel activation to be used. Can be either 'softmax' for softmax or 'exp' for exponential, for the self-attention layer.
- **lambda_attention** (*float*) – Constant to be multiplied with the attention matrix in the self-attention layer.
- **lambda_distance** (*float*) – Constant to be multiplied with the distance matrix in the self-attention layer.
- **h** (*int*) – Number of attention heads for the self-attention layer.
- **sa_hsize** (*int*) – Size of dense layer in the self-attention layer.
- **sa_dropout_p** (*float*) – Dropout probability for the self-attention layer.
- **output_bias** (*bool*) – If True, dense layers will use bias vectors in the self-attention layer.
- **d_input** (*int*) – Size of input layer in the feed-forward layer.
- **d_hidden** (*int*) – Size of hidden layer in the feed-forward layer.
- **d_output** (*int*) – Size of output layer in the feed-forward layer.
- **activation** (*str*) – Activation function to be used in the feed-forward layer. Can choose between 'relu' for ReLU, 'leakyrelu' for LeakyReLU, 'prelu' for PReLU, 'tanh' for TanH, 'selu' for SELU, 'elu' for ELU and 'linear' for linear activation.
- **n_layers** (*int*) – Number of layers in the feed-forward layer.
- **dropout_p** (*float*) – Dropout probability in the feeed-forward layer.
- **encoder_hsize** (*int*) – Size of Dense layer for the encoder itself.
- **encoder_dropout_p** (*float*) – Dropout probability for connections in the encoder layer.

forward(*x: Tensor*, *mask: Tensor*, *adj_matrix: Tensor*, *distance_matrix: Tensor*, *sa_dropout_p: float = 0.0*) → Tensor

Output computation for the MATEncoder layer.

In the MATEncoderLayer intialization, self.sublayer is defined as an nn.ModuleList of 2 layers. We will be passing our computation through these layers sequentially. nn.ModuleList is subscriptable and thus we can access it as self.sublayer[0], for example.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **mask** (*torch.Tensor*) – Masks out padding values so that they are not taken into account when computing the attention score.
- **adj_matrix** (*torch.Tensor*) – Adjacency matrix of a molecule.

- **distance_matrix** (*torch.Tensor*) – Distance matrix of a molecule.
- **sa_dropout_p** (*float*) – Dropout probability for the self-attention layer (MultiHeadedMATAttention).

class MultiHeadedMATAttention(*dist_kernel: str = 'softmax'*, *lambda_attention: float = 0.33*, *lambda_distance: float = 0.33*, *h: int = 16*, *hsize: int = 1024*, *dropout_p: float = 0.0*, *output_bias: bool = True*)

First constructs an attention layer tailored to the Molecular Attention Transformer [1]_ and then converts it into Multi-Headed Attention.

In Multi-Headed attention the attention mechanism multiple times parallely through the multiple attention heads. Thus, different subsequences of a given sequences can be processed differently. The query, key and value parameters are split multiple ways and each split is passed separately through a different attention head. .. rubric:: References

Examples

```
>>> from deepchem.models.torch_models.layers import MultiHeadedMATAttention,␣
↪MATEmbedding
>>> import deepchem as dc
>>> import torch
>>> input_smile = "CC"
>>> feat = dc.feat.MATFeaturizer()
>>> input_smile = "CC"
>>> out = feat.featurize(input_smile)
>>> node = torch.tensor(out[0].node_features).float().unsqueeze(0)
>>> adj = torch.tensor(out[0].adjacency_matrix).float().unsqueeze(0)
>>> dist = torch.tensor(out[0].distance_matrix).float().unsqueeze(0)
>>> mask = torch.sum(torch.abs(node), dim=-1) != 0
>>> layer = MultiHeadedMATAttention(
...    dist_kernel='softmax',
...    lambda_attention=0.33,
...    lambda_distance=0.33,
...    h=16,
...    hsize=1024,
...    dropout_p=0.0)
>>> op = MATEmbedding()(node)
>>> output = layer(op, op, op, mask, adj, dist)
```

__init__(*dist_kernel: str = 'softmax'*, *lambda_attention: float = 0.33*, *lambda_distance: float = 0.33*, *h: int = 16*, *hsize: int = 1024*, *dropout_p: float = 0.0*, *output_bias: bool = True*)

Initialize a multi-headed attention layer. :param dist_kernel: Kernel activation to be used. Can be either 'softmax' for softmax or 'exp' for exponential. :type dist_kernel: str :param lambda_attention: Constant to be multiplied with the attention matrix. :type lambda_attention: float :param lambda_distance: Constant to be multiplied with the distance matrix. :type lambda_distance: float :param h: Number of attention heads. :type h: int :param hsize: Size of dense layer. :type hsize: int :param dropout_p: Dropout probability. :type dropout_p: float :param output_bias: If True, dense layers will use bias vectors. :type output_bias: bool

forward(*query: Tensor*, *key: Tensor*, *value: Tensor*, *mask: Tensor*, *adj_matrix: Tensor*, *distance_matrix: Tensor*, *dropout_p: float = 0.0*, *eps: float = 1e-06*, *inf: float = 1000000000000.0*) → Tensor

Output computation for the MultiHeadedAttention layer. :param query: Standard query parameter for attention. :type query: torch.Tensor :param key: Standard key parameter for attention. :type key: torch.Tensor :param value: Standard value parameter for attention. :type value: torch.Tensor :param mask: Masks out padding values so that they are not taken into account when computing the attention score. :type mask: torch.Tensor :param adj_matrix: Adjacency matrix of the input molecule, returned from dc.feat.MATFeaturizer() :type adj_matrix: torch.Tensor :param dist_matrix: Distance matrix of the input molecule, returned from dc.feat.MATFeaturizer() :type dist_matrix: torch.Tensor :param dropout_p: Dropout probability. :type dropout_p: float :param eps: Epsilon value :type eps: float :param inf: Value of infinity to be used. :type inf: float

class SublayerConnection(*size: int*, *dropout_p: float = 0.0*)

SublayerConnection layer based on the paper Attention Is All You Need.

The SublayerConnection normalizes and adds dropout to output tensor of an arbitary layer. It further adds a residual layer connection between the input of the arbitary layer and the dropout-adjusted layer output.

Examples

```
>>> from deepchem.models.torch_models.layers import SublayerConnection
>>> scale = 0.35
>>> layer = SublayerConnection(2, 0.)
>>> input_ar = torch.tensor([[1., 2.], [5., 6.]])
>>> output = layer(input_ar, input_ar)
```

__init__(*size: int*, *dropout_p: float = 0.0*)

Initialize a SublayerConnection Layer.

Parameters

- **size** (*int*) – Size of layer.
- **dropout_p** (*float*) – Dropout probability.

forward(*x: Tensor*, *output: Tensor*) → Tensor

Output computation for the SublayerConnection layer.

Takes an input tensor x, then adds the dropout-adjusted sublayer output for normalized x to it. This is done to add a residual connection followed by LayerNorm.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **output** (*torch.Tensor*) – Layer whose normalized output will be added to x.

class PositionwiseFeedForward(*d_input: int = 1024*, *d_hidden: int = 1024*, *d_output: int = 1024*, *activation: str = 'leakyrelu'*, *n_layers: int = 1*, *dropout_p: float = 0.0*, *dropout_at_input_no_act: bool = False*)

PositionwiseFeedForward is a layer used to define the position-wise feed-forward (FFN) algorithm for the Molecular Attention Transformer [1]_

Each layer in the MAT encoder contains a fully connected feed-forward network which applies two linear transformations and the given activation function. This is done in addition to the SublayerConnection module.

Note: This modified version of *PositionwiseFeedForward* class contains *dropout_at_input_no_act* condition to facilitate its use in defining
the feed-forward (FFN) algorithm for the Directed Message Passing Neural Network (D-MPNN) [2]_

References

Examples

```
>>> from deepchem.models.torch_models.layers import PositionwiseFeedForward
>>> feed_fwd_layer = PositionwiseFeedForward(d_input = 2, d_hidden = 2, d_output =␣
↪2, activation = 'relu', n_layers = 1, dropout_p = 0.1)
>>> input_tensor = torch.tensor([[1., 2.], [5., 6.]])
>>> output_tensor = feed_fwd_layer(input_tensor)
```

`__init__`(*d_input: int = 1024*, *d_hidden: int = 1024*, *d_output: int = 1024*, *activation: str = 'leakyrelu'*, *n_layers: int = 1*, *dropout_p: float = 0.0*, *dropout_at_input_no_act: bool = False*)

Initialize a PositionwiseFeedForward layer.

Parameters

- **`d_input`** (*`int`*) – Size of input layer.
- **`d_hidden`** (*`int (same as d_input if d_output = 0)`*) – Size of hidden layer.
- **`d_output`** (*`int (same as d_input if d_output = 0)`*) – Size of output layer.
- **`activation`** (*`str`*) – Activation function to be used. Can choose between 'relu' for ReLU, 'leakyrelu' for LeakyReLU, 'prelu' for PReLU, 'tanh' for TanH, 'selu' for SELU, 'elu' for ELU and 'linear' for linear activation.
- **`n_layers`** (*`int`*) – Number of layers.
- **`dropout_p`** (*`float`*) – Dropout probability.
- **`dropout_at_input_no_act`** (*`bool`*) – If true, dropout is applied on the input tensor. For single layer, it is not passed to an activation function.

`forward`(*x: Tensor*) → Tensor

Output Computation for the PositionwiseFeedForward layer.

Parameters

`x` (*`torch.Tensor`*) – Input tensor.

`class` **`MATEmbedding`**(*d_input: int = 36*, *d_output: int = 1024*, *dropout_p: float = 0.0*)

Embedding layer to create embedding for inputs.

In an embedding layer, input is taken and converted to a vector representation for each input. In the MATEmbedding layer, an input tensor is processed through a dropout-adjusted linear layer and the resultant vector is returned.

References

Examples

```
>>> from deepchem.models.torch_models.layers import MATEmbedding
>>> layer = MATEmbedding(d_input = 3, d_output = 3, dropout_p = 0.2)
>>> input_tensor = torch.tensor([1., 2., 3.])
>>> output = layer(input_tensor)
```

`__init__`(*d_input: int = 36*, *d_output: int = 1024*, *dropout_p: float = 0.0*)

Initialize a MATEmbedding layer.

Parameters

- **d_input** (*int*) – Size of input layer.
- **d_output** (*int*) – Size of output layer.
- **dropout_p** (*float*) – Dropout probability for layer.

forward(*x: Tensor*) → Tensor

Computation for the MATEmbedding layer.

Parameters

x (*torch.Tensor*) – Input tensor to be converted into a vector.

class MATGenerator(*hsize: int = 1024*, *aggregation_type: str = 'mean'*, *d_output: int = 1*, *n_layers: int = 1*, *dropout_p: float = 0.0*, *attn_hidden: int = 128*, *attn_out: int = 4*)

MATGenerator defines the linear and softmax generator step for the Molecular Attention Transformer [1]_.

In the MATGenerator, a Generator is defined which performs the Linear + Softmax generation step. Depending on the type of aggregation selected, the attention output layer performs different operations.

References

Examples

```
>>> from deepchem.models.torch_models.layers import MATGenerator
>>> layer = MATGenerator(hsize = 3, aggregation_type = 'mean', d_output = 1, n_
↪layers = 1, dropout_p = 0.3, attn_hidden = 128, attn_out = 4)
>>> input_tensor = torch.tensor([1., 2., 3.])
>>> mask = torch.tensor([1., 1., 1.])
>>> output = layer(input_tensor, mask)
```

__init__(*hsize: int = 1024*, *aggregation_type: str = 'mean'*, *d_output: int = 1*, *n_layers: int = 1*, *dropout_p: float = 0.0*, *attn_hidden: int = 128*, *attn_out: int = 4*)

Initialize a MATGenerator.

Parameters

- **hsize** (*int*) – Size of input layer.
- **aggregation_type** (*str*) – Type of aggregation to be used. Can be 'grover', 'mean' or 'contextual'.
- **d_output** (*int*) – Size of output layer.
- **n_layers** (*int*) – Number of layers in MATGenerator.
- **dropout_p** (*float*) – Dropout probability for layer.
- **attn_hidden** (*int*) – Size of hidden attention layer.
- **attn_out** (*int*) – Size of output attention layer.

forward(*x: Tensor*, *mask: Tensor*) → Tensor

Computation for the MATGenerator layer.

Parameters

- **x** (*torch.Tensor*) – Input tensor.

- **mask** (*torch.Tensor*) – Mask for padding so that padded values do not get included in attention score calculation.

cosine_dist(*x*, *y*)

Computes the inner product (cosine similarity) between two tensors.

This assumes that the two input tensors contain rows of vectors where each column represents a different feature. The output tensor will have elements that represent the inner product between pairs of normalized vectors in the rows of *x* and *y*. The two tensors need to have the same number of columns, because one cannot take the dot product between vectors of different lengths. For example, in sentence similarity and sentence classification tasks, the number of columns is the embedding size. In these tasks, the rows of the input tensors would be different test vectors or sentences. The input tensors themselves could be different batches. Using vectors or tensors of all 0s should be avoided.

The vectors in the input tensors are first l2-normalized such that each vector

has length or magnitude of 1. The inner product (dot product) is then taken

between corresponding pairs of row vectors in the input tensors and returned.

Examples

The cosine similarity between two equivalent vectors will be 1. The cosine similarity between two equivalent tensors (tensors where all the elements are the same) will be a tensor of 1s. In this scenario, if the input tensors *x* and *y* are each of shape *(n,p)*, where each element in *x* and *y* is the same, then the output tensor would be a tensor of shape *(n,n)* with 1 in every entry.

```
>>> import numpy as np
>>> import tensorflow as tf
>>> import deepchem.models.layers as layers
>>> x = tf.ones((6, 4), dtype=tf.dtypes.float32, name=None)
>>> y_same = tf.ones((6, 4), dtype=tf.dtypes.float32, name=None)
>>> cos_sim_same = layers.cosine_dist(x,y_same)
```

x and *y_same* are the same tensor (equivalent at every element, in this case 1). As such, the pairwise inner product of the rows in *x* and *y* will always be 1. The output tensor will be of shape (6,6).

```
>>> diff = cos_sim_same - tf.ones((6, 6), dtype=tf.dtypes.float32, name=None)
>>> np.allclose(0.0, tf.reduce_sum(diff).numpy(), atol=1e-05)
True
>>> cos_sim_same.shape
TensorShape([6, 6])
```

The cosine similarity between two orthogonal vectors will be 0 (by definition). If every row in *x* is orthogonal to every row in *y*, then the output will be a tensor of 0s. In the following example, each row in the tensor *x1* is orthogonal to each row in *x2* because they are halves of an identity matrix.

```
>>> identity_tensor = tf.eye(512, dtype=tf.dtypes.float32)
>>> x1 = identity_tensor[0:256,:]
>>> x2 = identity_tensor[256:512,:]
>>> cos_sim_orth = layers.cosine_dist(x1,x2)
```

Each row in *x1* is orthogonal to each row in *x2*. As such, the pairwise inner product of the rows in *x1`and `x2* will always be 0. Furthermore, because the shape of the input tensors are both of shape *(256,512)*, the output tensor will be of shape *(256,256)*.

```
>>> np.allclose(0.0, tf.reduce_sum(cos_sim_orth).numpy(), atol=1e-05)
True
>>> cos_sim_orth.shape
TensorShape([256, 256])
```

Parameters

- **x** (*tf.Tensor*) – Input Tensor of shape *(n, p)*. The shape of this input tensor should be *n* rows by *p* columns. Note that *n* need not equal *m* (the number of rows in *y*).
- **y** (*tf.Tensor*) – Input Tensor of shape *(m, p)* The shape of this input tensor should be *m* rows by *p* columns. Note that *m* need not equal *n* (the number of rows in *x*).

Returns

Returns a tensor of shape *(n, m)*, that is, *n* rows by *m* columns. Each *i,j*-th entry of this output tensor is the inner product between the l2-normalized *i*-th row of the input tensor *x* and the the l2-normalized *j*-th row of the output tensor *y*.

Return type

tf.Tensor

class GraphNetwork(*n_node_features: int = 32*, *n_edge_features: int = 32*, *n_global_features: int = 32*, *is_undirected: bool = True*, *residual_connection: bool = True*)

Graph Networks

A Graph Network [1]_ takes a graph as input and returns an updated graph as output. The output graph has same structure as input graph but it has updated node features, edge features and global state features.

Parameters

- **n_node_features** (*int*) – Number of features in a node
- **n_edge_features** (*int*) – Number of features in a edge
- **n_global_features** (*int*) – Number of global features
- **is_undirected** (*bool, optional (default True)*) – Directed or undirected graph
- **residual_connection** (*bool, optional (default True)*) – If True, the layer uses a residual connection during training

Example

```
>>> import torch
>>> from deepchem.models.torch_models.layers import GraphNetwork as GN
>>> n_nodes, n_node_features = 5, 10
>>> n_edges, n_edge_features = 5, 2
>>> n_global_features = 4
>>> node_features = torch.randn(n_nodes, n_node_features)
>>> edge_features = torch.randn(n_edges, n_edge_features)
>>> edge_index = torch.tensor([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]).long()
>>> global_features = torch.randn(1, n_global_features)
>>> gn = GN(n_node_features=n_node_features, n_edge_features=n_edge_features, n_
↪global_features=n_global_features)
>>> node_features, edge_features, global_features = gn(node_features, edge_index,␣
↪edge_features, global_features)
```

References

__init__(*n_node_features: int = 32*, *n_edge_features: int = 32*, *n_global_features: int = 32*, *is_undirected: bool = True*, *residual_connection: bool = True*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*node_features: Tensor*, *edge_index: Tensor*, *edge_features: Tensor*, *global_features: Tensor*, *batch: Tensor | None = None*) → Tuple[Tensor, Tensor, Tensor]

Output computation for a GraphNetwork

Parameters

- **node_features** (*torch.Tensor*) – Input node features of shape $(|\mathcal{V}|, F_n)$
- **edge_index** (*torch.Tensor*) – Edge indexes of shape $(2, |\mathcal{E}|)$
- **edge_features** (*torch.Tensor*) – Edge features of the graph, shape: $(|\mathcal{E}|, F_e)$
- **global_features** (*torch.Tensor*) – Global features of the graph, shape: $(F_g, 1)$ where, $|\mathcal{V}|$ and $|\mathcal{E}|$ denotes the number of nodes and edges in the graph, F_n, F_e, F_g denotes the number of node features, edge features and global state features respectively.
- **batch** (*torch.LongTensor (optional, default: None)*) – A vector that maps each node to its respective graph identifier. The attribute is used only when more than one graph are batched together during a single forward pass.

class Affine(*dim: int*)

Class which performs the Affine transformation.

This transformation is based on the affinity of the base distribution with the target distribution. A geometric transformation is applied where the parameters performs changes on the scale and shift of a function (inputs).

Normalizing Flow transformations must be bijective in order to compute the logarithm of jacobian's determinant. For this reason, transformations must perform a forward and inverse pass.

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.layers import Affine
>>> import torch
>>> from torch.distributions import MultivariateNormal
>>> # initialize the transformation layer's parameters
>>> dim = 2
>>> samples = 96
>>> transforms = Affine(dim)
>>> # forward pass based on a given distribution
>>> distribution = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> input = distribution.sample(torch.Size((samples, dim)))
>>> len(transforms.forward(input))
2
>>> # inverse pass based on a distribution
>>> len(transforms.inverse(input))
2
```

__init__(*dim: int*) → None

Create a Affine transform layer.

Parameters
dim (*int*) – Value of the Nth dimension of the dataset.

forward(*x: Tensor*) → Tuple[Tensor, Tensor]

Performs a transformation between two different distributions. This particular transformation represents the following function:

$$y = x * exp(a) + b$$

where a is scale parameter and b performs a shift. This class also returns the logarithm of the jacobians determinant which is useful when invert a transformation and compute the probability of the transformation.

Parameters
x (*torch.Tensor*) – Tensor sample with the initial distribution data which will pass into the normalizing flow algorithm.

Returns

- **y** (*torch.Tensor*) – Transformed tensor according to Affine layer with the shape of 'x'.
- **log_det_jacobian** (*torch.Tensor*) – Tensor which represents the info about the deviation of the initial and target distribution.

inverse(*y: Tensor*) → Tuple[Tensor, Tensor]

Performs a transformation between two different distributions. This transformation represents the bacward pass of the function mention before. Its mathematical representation is x = (y - b) / exp(a) , where "a" is scale parameter and "b" performs a shift. This class also returns the logarithm of the jacobians determinant which is useful when invert a transformation and compute the probability of the transformation.

Parameters
y (*torch.Tensor*) – Tensor sample with transformed distribution data which will be used in the normalizing algorithm inverse pass.

Returns

- **x** (*torch.Tensor*) – Transformed tensor according to Affine layer with the shape of 'y'.
- **inverse_log_det_jacobian** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

class RealNVPLayer(*mask: Tensor*, *hidden_size: int*)

Real NVP Transformation Layer

This class class is a constructor transformation layer used on a NormalizingFLow model. The Real Non-Preserving-Volumen (Real NVP) is a type of normalizing flow layer which gives advantages over this mainly because an ease to compute the inverse pass [realnvp1], this is to learn a target distribution.

Example

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> from torch.distributions import MultivariateNormal
>>> from deepchem.models.torch_models.layers import RealNVPLayer
>>> dim = 2
>>> samples = 96
>>> data = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> tensor = data.sample(torch.Size((samples, dim)))
>>> layers = 4
>>> hidden_size = 16
>>> masks = F.one_hot(torch.tensor([i % 2 for i in range(layers)])).float()
>>> layers = nn.ModuleList([RealNVPLayer(mask, hidden_size) for mask in masks])
```

```
>>> for layer in layers:
...   _, inverse_log_det_jacobian = layer.inverse(tensor)
...   inverse_log_det_jacobian = inverse_log_det_jacobian.detach().numpy()
>>> len(inverse_log_det_jacobian)
96
```

References

Distributions of Normalizing Flows. (2017). Retrieved from http://arxiv.org/abs/2110.15828

`__init__`(*mask: Tensor*, *hidden_size: int*) → None

Parameters

- **`mask`** (*`torch.Tensor`*) – Tensor with zeros and ones and its size depende on the number of layers and dimenssions the user request.
- **`hidden_size`** (*`int`*) – The size of the outputs and inputs used on the internal nodes of the transformation layer.

`forward`(*x: Sequence*) → Tuple[Tensor, Tensor]

Forward pass.

This particular transformation is represented by the following function: y = x + (1 - x) * exp(s(x)) + t(x), where t and s needs an activation function. This class also returns the logarithm of the jacobians determinant which is useful when invert a transformation and compute the probability of the transformation.

Parameters

`x` (*`Sequence`*) – Tensor sample with the initial distribution data which will pass into the normalizing algorithm

Returns

- **y** (*torch.Tensor*) – Transformed tensor according to Real NVP layer with the shape of 'x'.
- **log_det_jacobian** (*torch.Tensor*) – Tensor which represents the info about the deviation of the initial and target distribution.

`inverse`(*y: Sequence*) → Tuple[Tensor, Tensor]

Inverse pass

This class performs the inverse of the previous method (formward). Also, this metehod returns the logarithm of the jacobians determinant which is useful to compute the learneable features of target distribution.

Parameters

`y` (*`Sequence`*) – Tensor sample with transformed distribution data which will be used in the normalizing algorithm inverse pass.

Returns

- **x** (*torch.Tensor*) – Transformed tensor according to Real NVP layer with the shape of 'y'.
- **inverse_log_det_jacobian** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

class DMPNNEncoderLayer(*use_default_fdim: bool = True*, *atom_fdim: int = 133*, *bond_fdim: int = 14*, *d_hidden: int = 300*, *depth: int = 3*, *bias: bool = False*, *activation: str = 'relu'*, *dropout_p: float = 0.0*, *aggregation: str = 'mean'*, *aggregation_norm: int | float = 100*)

Encoder layer for use in the Directed Message Passing Neural Network (D-MPNN) [1]_.

The role of the DMPNNEncoderLayer class is to generate molecule encodings in following steps:

- Message passing phase
- Get new atom hidden states and readout phase
- Concatenate the global features

Let the diagram given below represent a molecule containing 5 atoms (nodes) and 4 bonds (edges):-

1 — 5
|
2 — 4
|
3

Let the bonds from atoms 1->2 (**B[12]**) and 2->1 (**B[21]**) be considered as 2 different bonds. Hence, by considering the same for all atoms, the total number of bonds = 8.

Let:

- **atom features** : `a1, a2, a3, a4, a5`
- **hidden states of atoms** : `h1, h2, h3, h4, h5`
- **bond features bonds** : `b12, b21, b23, b32, b24, b42, b15, b51`
- **initial hidden states of bonds** : `(0)h12, (0)h21, (0)h23, (0)h32, (0)h24, (0)h42, (0)h15, (0)h51`

The hidden state of every bond is a function of the concatenated feature vector which contains concatenation of the **features of initial atom of the bond** and **bond features**.

Example: `(0)h21 = func1(concat(a2, b21))`

Note: Here func1 is `self.W_i`

The Message passing phase

The goal of the message-passing phase is to generate **hidden states of all the atoms in the molecule**.

The hidden state of an atom is **a function of concatenation of atom features and messages (at T depth)**.

A message is a sum of **hidden states of bonds coming to the atom (at T depth)**.

Note: Depth refers to the number of iterations in the message passing phase (here, T iterations). After each iteration, the hidden states of the bonds are updated.

Example: `h1 = func3(concat(a1, m1))`

Note: Here func3 is `self.W_o`.

m1 refers to the message coming to the atom.

`m1 = (T-1)h21 + (T-1)h51` (hidden state of bond 2->1 + hidden state of bond 5->1) (at T depth)

for, depth T = 2:

- the hidden states of the bonds @ 1st iteration will be => (0)h21, (0)h51
- the hidden states of the bonds @ 2nd iteration will be => (1)h21, (1)h51

The hidden states of the bonds in 1st iteration are already know. For hidden states of the bonds in 2nd iteration, we follow the criterion that:

- hidden state of the bond is a function of **initial hidden state of bond**

and **messages coming to that bond in that iteration**

Example: `(1)h21 = func2( (0)h21 , (1)m21 )`

Note: Here func2 is `self.W_h`.

(1)m21 refers to the messages coming to that bond 2->1 in that 2nd iteration.

Messages coming to a bond in an iteration is **a sum of hidden states of bonds (from previous iteration) coming to this bond**.

Example: `(1)m21 = (0)h32 + (0)h42`

2 <— 3
^
|
4

Computing the messages

```
                      B0      B1      B2      B3      B4      B5      B6      B7     ␣
↪ B8
f_ini_atoms_bonds = [(0)h12, (0)h21, (0)h23, (0)h32, (0)h24, (0)h42, (0)h15, (0)h51,
↪ h(-1)]
```

Note: h(-1) is an empty array of the same size as other hidden states of bond states.

```
             B0      B1      B2      B3      B4      B5      B6      B7       B8
mapping = [ [-1,B7] [B3,B5] [B0,B5] [-1,-1] [B0,B3] [-1,-1] [B1,-1] [-1,-1]  [-1,-
↪1] ]
```

Later, the encoder will map the concatenated features from the `f_ini_atoms_bonds` to `mapping` in each iteration upto Tth iteration.

Next the encoder will sum-up the concat features within same bond index.

```
                (1)m12          (1)m21          (1)m23              (1)m32      ␣
↪  (1)m24           (1)m42          (1)m15          (1)m51              m(-1)
message = [ [h(-1) + (0)h51] [(0)h32 + (0)h42] [(0)h12 + (0)h42] [h(-1) + h(-1)]␣
↪[(0)h12 + (0)h32] [h(-1) + h(-1)] [(0)h21 + h(-1)] [h(-1) + h(-1)]  [h(-1) + h(-
↪1)] ]
```

Hence, this is how encoder can get messages for message-passing steps.

Get new atom hidden states and readout phase

Hence now for h1:

```
h1 = func3(
            concat(
                 a1,
                 [
                    func2( (0)h21 , (0)h32 + (0)h42 ) +
                    func2( (0)h51 , 0               )
                 ]
                )
         )
```

Similarly, h2, h3, h4 and h5 are calculated.

Next, all atom hidden states are concatenated to make a feature vector of the molecule:

`mol_encodings = [[h1, h2, h3, h4, h5]]`

Concatenate the global features

Let, `global_features = [[gf1, gf2, gf3]]` This array contains molecule level features. In case of this example, it contains 3 global features.

Hence after concatenation,

`mol_encodings = [[h1, h2, h3, h4, h5, gf1, gf2, gf3]]` (Final output of the encoder)

References

Examples

```
>>> from rdkit import Chem
>>> import torch
>>> import deepchem as dc
>>> input_smile = "CC"
>>> feat = dc.feat.DMPNNFeaturizer(features_generators=['morgan'])
>>> graph = feat.featurize(input_smile)
>>> from deepchem.models.torch_models.dmpnn import _MapperDMPNN
>>> mapper = _MapperDMPNN(graph[0])
>>> atom_features, f_ini_atoms_bonds, atom_to_incoming_bonds, mapping, global_
↪features = mapper.values
>>> atom_features = torch.from_numpy(atom_features).float()
>>> f_ini_atoms_bonds = torch.from_numpy(f_ini_atoms_bonds).float()
>>> atom_to_incoming_bonds = torch.from_numpy(atom_to_incoming_bonds)
>>> mapping = torch.from_numpy(mapping)
>>> global_features = torch.from_numpy(global_features).float()
```

(continues on next page)

(continued from previous page)

```
>>> molecules_unbatch_key = len(atom_features)
>>> layer = DMPNNEncoderLayer(d_hidden=2)
>>> output = layer(atom_features, f_ini_atoms_bonds, atom_to_incoming_bonds,␣
↪mapping, global_features, molecules_unbatch_key)
```

`__init__`(*use_default_fdim: bool = True*, *atom_fdim: int = 133*, *bond_fdim: int = 14*, *d_hidden: int = 300*, *depth: int = 3*, *bias: bool = False*, *activation: str = 'relu'*, *dropout_p: float = 0.0*, *aggregation: str = 'mean'*, *aggregation_norm: int | float = 100*)

Initialize a DMPNNEncoderLayer layer.

Parameters

- **`use_default_fdim`** (*`bool`*) – If `True`, `self.atom_fdim` and `self.bond_fdim` are initialized using values from the GraphConvConstants class. If `False`, `self.atom_fdim` and `self.bond_fdim` are initialized from the values provided.
- **`atom_fdim`** (*`int`*) – Dimension of atom feature vector.
- **`bond_fdim`** (*`int`*) – Dimension of bond feature vector.
- **`d_hidden`** (*`int`*) – Size of hidden layer in the encoder layer.
- **`depth`** (*`int`*) – No of message passing steps.
- **`bias`** (*`bool`*) – If `True`, dense layers will use bias vectors.
- **`activation`** (*`str`*) – Activation function to be used in the encoder layer. Can choose between ‘relu’ for ReLU, ‘leakyrelu’ for LeakyReLU, ‘prelu’ for PReLU, ‘tanh’ for TanH, ‘selu’ for SELU, and ‘elu’ for ELU.
- **`dropout_p`** (*`float`*) – Dropout probability for the encoder layer.
- **`aggregation`** (*`str`*) – Aggregation type to be used in the encoder layer. Can choose between ‘mean’, ‘sum’, and ‘norm’.
- **`aggregation_norm`** (*`Union[int, float]`*) – Value required if *aggregation* type is ‘norm’.

`forward`(*atom_features: Tensor*, *f_ini_atoms_bonds: Tensor*, *atom_to_incoming_bonds: Tensor*, *mapping: Tensor*, *global_features: Tensor*, *molecules_unbatch_key: List*) → Tensor

Output computation for the DMPNNEncoderLayer.

Steps:

- Get original bond hidden states from concatenation of initial atom and bond features. (`input`)
- Get initial messages hidden states. (`message`)
- Execute message passing step for `self.depth - 1` iterations.
- Get atom hidden states using atom features and message hidden states.
- Get molecule encodings.
- Concatenate global molecular features and molecule encodings.

Parameters

- **`atom_features`** (*`torch.Tensor`*) – Tensor containing atoms features.
- **`f_ini_atoms_bonds`** (*`torch.Tensor`*) – Tensor containing concatenated feature vector which contains concatenation of initial atom and bond features.

- **atom_to_incoming_bonds** (*torch.Tensor*) – Tensor containing mapping from atom index to list of indicies of incoming bonds.
- **mapping** (*torch.Tensor*) – Tensor containing the mapping that maps bond index to 'array of indices of the bonds' incoming at the initial atom of the bond (excluding the reverse bonds).
- **global_features** (*torch.Tensor*) – Tensor containing molecule features.
- **molecules_unbatch_key** (*List*) – List containing number of atoms in various molecules of a batch

Returns
output – Tensor containing the encodings of the molecules.

Return type
torch.Tensor

class **InfoGraphEncoder**(*num_features*, *edge_features*, *embedding_dim*)

The encoder for the InfoGraph model. It is a message passing graph convolutional network that produces encoded representations for molecular graph inputs.

Parameters

- **num_features** (*int*) – Number of node features for each input
- **edge_features** (*int*) – Number of edge features for each input
- **embedding_dim** (*int*) – Dimension of the embedding

Example

```
>>> import numpy as np
>>> from deepchem.models.torch_models.infograph import InfoGraphEncoder
>>> from deepchem.feat.graph_data import GraphData
>>> encoder = InfoGraphEncoder(num_features=25, edge_features=10, embedding_dim=32)
>>> node_features = np.random.randn(10, 25)
>>> edge_index = np.array([[0, 1, 2], [1, 2, 3]])
>>> edge_features = np.random.randn(3, 10)
>>> graph_index = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> data = GraphData(node_features=node_features, edge_index=edge_index, edge_
↪features=edge_features, graph_index=graph_index).numpy_to_torch()
>>> embedding, feature_map = encoder(data)
>>> print(embedding.shape)
torch.Size([1, 64])
```

__init__(*num_features*, *edge_features*, *embedding_dim*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*data*)

Encode input graphs into an embedding and feature map.

Parameters
data (*Union[BatchGraphData, GraphData]*) – Contains information about graphs.

Returns

- *torch.Tensor* – Encoded tensor of input data.

- *torch.Tensor* – Feature map tensor of input data.

class GINEncoder(*num_features: int*, *embedding_dim: int*, *num_gc_layers: int = 5*)

Graph Information Network (GIN) encoder. This is a graph convolutional network that produces encoded representations for molecular graph inputs.

Parameters

- **num_features** (*int*) – The number of node features
- **embedding_dim** (*int*) – The dimension of the output embedding
- **num_gc_layers** (*int, optional (default 5)*) – The number of graph convolutional layers to use

Example

```
>>> import numpy as np
>>> from deepchem.models.torch_models.infograph import GINEncoder
>>> from deepchem.feat.graph_data import GraphData
>>> encoder = GINEncoder(num_features=25, embedding_dim=32)
>>> node_features = np.random.randn(10, 25)
>>> edge_index = np.array([[0, 1, 2], [1, 2, 3]])
>>> edge_features = np.random.randn(3, 10)
>>> graph_index = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> data = GraphData(node_features=node_features, edge_index=edge_index, edge_
↪features=edge_features, graph_index=graph_index).numpy_to_torch()
>>> embedding, intermediate_embeddings = encoder(data)
>>> print(embedding.shape)
torch.Size([1, 30])
```

References

__init__(*num_features: int*, *embedding_dim: int*, *num_gc_layers: int = 5*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*data*)

Encodes the input graph data.

Parameters
data (*BatchGraphData*) – The batched input graph data.

Returns
A tuple containing the encoded representation and intermediate embeddings.

Return type
Tuple[torch.Tensor, torch.Tensor]

class SetGather(*M: int*, *batch_size: int*, *n_hidden: int = 100*, *init='orthogonal'*, ***kwargs*)

set2set gather layer for graph-based model

Models using this layer must set *pad_batches=True*

Torch Equivalent of Keras SetGather layer

Parameters

- **M** (*int*) – Number of LSTM steps

- **batch_size** (*int*) – Number of samples in a batch(all batches must have same size)
- **n_hidden** (*int, optional*) – number of hidden units in the passing phase

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> from deepchem.models.torch_models import layers
>>> total_n_atoms = 4
>>> n_atom_feat = 4
>>> atom_feat = np.random.rand(total_n_atoms, n_atom_feat)
>>> atom_split = np.array([0, 0, 1, 1], dtype=np.int32)
>>> gather = layers.SetGather(2, 2, n_hidden=4)
>>> output_molecules = gather([atom_feat, atom_split])
>>> print(output_molecules.shape)
torch.Size([2, 8])
```

__init__(*M: int*, *batch_size: int*, *n_hidden: int = 100*, *init='orthogonal'*, ***kwargs*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*inputs: List*) → Tensor

Perform M steps of set2set gather,

Detailed descriptions in: https://arxiv.org/abs/1511.06391

Parameters

inputs (*List*) – This contains two elements. atom_features: np.ndarray atom_split: np.ndarray

Returns

q_star – Final state of the model after all M steps.

Return type

torch.Tensor

class GNN(*node_type_embedding*, *chirality_embedding*, *gconvs*, *batch_norms*, *dropout*, *jump_knowledge*, *init_emb=False*)

GNN module for the GNNModular model.

This module is responsible for the graph neural network layers in the GNNModular model.

Parameters

- **node_type_embedding** (*torch.nn.Embedding*) – Embedding layer for node types.
- **chirality_embedding** (*torch.nn.Embedding*) – Embedding layer for chirality tags.
- **gconvs** (*torch.nn.ModuleList*) – ModuleList of graph convolutional layers.
- **batch_norms** (*torch.nn.ModuleList*) – ModuleList of batch normalization layers.
- **dropout** (*int*) – Dropout probability.
- **jump_knowledge** (*str*) – The type of jump knowledge to use. [1] Must be one of "last", "sum", "max", "concat" or "none". "last": Use the node representation from the last GNN layer. "concat": Concatenate the node representations from all GNN layers. "max": Take the element-wise maximum of the node representations from all GNN layers. "sum": Take the element-wise sum of the node representations from all GNN layers.

- **init_emb** (*bool*) – Whether to initialize the embedding layers with Xavier uniform initialization.

References

Example

```
>>> from deepchem.models.torch_models.gnn import GNNModular
>>> from deepchem.feat.graph_data import BatchGraphData
>>> from deepchem.feat.molecule_featurizers import SNAPFeaturizer
>>> featurizer = SNAPFeaturizer()
>>> smiles = ["C1=CC=CC=C1", "C1=CC=CC=C1C=O", "C1=CC=CC=C1C(=O)O"]
>>> features = featurizer.featurize(smiles)
>>> modular = GNNModular(emb_dim = 8, task = "edge_pred")
>>> batched_graph = BatchGraphData(features).numpy_to_torch(device=modular.device)
>>> gnnmodel = modular.gnn
>>> print(gnnmodel(batched_graph)[0].shape)
torch.Size([23, 8])
```

__init__(*node_type_embedding*, *chirality_embedding*, *gconvs*, *batch_norms*, *dropout*, *jump_knowledge*, *init_emb=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*data: BatchGraphData*)

Forward pass for the GNN module.

Parameters
data (*BatchGraphData*) – Batched graph data.

class GNNHead(*pool*, *head*, *task*, *num_tasks*, *num_classes*)

Prediction head module for the GNNModular model.

Parameters

- **pool** (*Union[function,torch.nn.Module]*) – Pooling function or nn.Module to use
- **head** (*torch.nn.Module*) – Prediction head to use
- **task** (*str*) – The type of task. Must be one of “regression”, “classification”.
- **num_tasks** (*int*) – Number of tasks.
- **num_classes** (*int*) – Number of classes for classification.

__init__(*pool*, *head*, *task*, *num_tasks*, *num_classes*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*data*)

Forward pass for the GNN head module.

Parameters
data (*tuple*) – A tuple containing the node representations and the input graph data. node_representation is a torch.Tensor created after passing input through the GNN layers. input_batch is the original input BatchGraphData.

class LocalGlobalDiscriminator(*hidden_dim*)

This discriminator module is a linear layer without bias, used to measure the similarity between local node representations (*x*) and global graph representations (*summary*).

The goal of the discriminator is to distinguish between positive and negative pairs of local and global representations.

Examples

```
>>> import torch
>>> from deepchem.models.torch_models.gnn import LocalGlobalDiscriminator
>>> discriminator = LocalGlobalDiscriminator(hidden_dim=64)
>>> x = torch.randn(32, 64)  # Local node representations
>>> summary = torch.randn(32, 64)  # Global graph representations
>>> similarity_scores = discriminator(x, summary)
>>> print(similarity_scores.shape)
torch.Size([32])
```

__init__(*hidden_dim*)

self.weight is a learnable weight matrix of shape *(hidden_dim, hidden_dim)*.

nn.Parameters are tensors that require gradients and are optimized during the training process.

Parameters

hidden_dim (*int*) – The size of the hidden dimension for the weight matrix.

forward(*x*, *summary*)

Computes the product of *summary* and *self.weight*, and then calculates the element-wise product of *x* and the resulting matrix *h*. It then sums over the *hidden_dim* dimension, resulting in a tensor of shape *(batch_size,)*, which represents the similarity scores between the local and global representations.

Parameters

- **x** (*torch.Tensor*) – Local node representations of shape *(batch_size, hidden_dim)*.
- **summary** (*torch.Tensor*) – Global graph representations of shape *(batch_size, hidden_dim)*.

Returns

A tensor of shape *(batch_size,)*, representing the similarity scores between the local and global representations.

Return type

torch.Tensor

class AtomEncoder(*emb_dim*, *padding=False*)

Encodes atom features into embeddings based on the Open Graph Benchmark feature set in conformer_featurizer.

Parameters

- **emb_dim** (*int*) – The dimension that the returned embedding will have.
- **padding** (*bool, optional (default=False)*) – If true then the last index will be used for padding.

Examples

```
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import full_atom_
↪feature_dims
>>> atom_encoder = AtomEncoder(emb_dim=32)
>>> num_rows = 10
>>> atom_features = torch.stack([
... torch.randint(low=0, high=dim, size=(num_rows,))
... for dim in full_atom_feature_dims
... ], dim=1)
>>> atom_embeddings = atom_encoder(atom_features)
```

`__init__`(*emb_dim*, *padding=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`reset_parameters`()

Reset the parameters of the atom embeddings.

This method resets the weights of the atom embeddings by initializing them with a uniform distribution between -sqrt(3) and sqrt(3).

`forward`(*x*)

Compute the atom embeddings for the given atom features.

Parameters

x (*torch.Tensor, shape (batch_size, num_atoms, num_features)*) – The input atom features tensor.

Returns

x_embedding – The computed atom embeddings.

Return type

torch.Tensor, shape (batch_size, num_atoms, emb_dim)

class BondEncoder(*emb_dim*, *padding=False*)

Encodes bond features into embeddings based on the Open Graph Benchmark feature set in conformer_featurizer.

Parameters

- **emb_dim** (*int*) – The dimension that the returned embedding will have.
- **padding** (*bool, optional (default=False)*) – If true then the last index will be used for padding.

Examples

```
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import full_bond_
↪feature_dims
>>> bond_encoder = BondEncoder(emb_dim=32)
>>> num_rows = 10
>>> bond_features = torch.stack([
... torch.randint(low=0, high=dim, size=(num_rows,))
... for dim in full_bond_feature_dims
... ], dim=1)
>>> bond_embeddings = bond_encoder(bond_features)
```

`__init__`(*emb_dim*, *padding=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*edge_attr*)

Compute the bond embeddings for the given bond features.

Parameters

`edge_attr` (*torch.Tensor, shape (batch_size, num_edges, num_features)*) – The input bond features tensor.

Returns

bond_embedding – The computed bond embeddings.

Return type

torch.Tensor, shape (batch_size, num_edges, emb_dim)

class `PNALayer`(*in_dim: int*, *out_dim: int*, *in_dim_edges: int*, *aggregators: List[str]*, *scalers: List[str]*, *activation: Callable | str = 'relu'*, *dropout: float = 0.0*, *residual: bool = True*, *pairwise_distances: bool = False*, *batch_norm: bool = True*, *batch_norm_momentum=0.1*, *avg_d: Dict[str, float] = {'log': 1.0}*, *posttrans_layers: int = 2*, *pretrans_layers: int = 1*)

Principal Neighbourhood Aggregation Layer (PNA) from [1].

Parameters

- `in_dim` (*int*) – Input dimension of the node features.
- `out_dim` (*int*) – Output dimension of the node features.
- `in_dim_edges` (*int*) – Input dimension of the edge features.
- `aggregators` (*List[str]*) – List of aggregator functions to use. Options are "mean", "sum", "max", "min", "std", "var", "moment3", "moment4", "moment5".
- `scalers` (*List[str]*) – List of scaler functions to use. Options are "identity", "amplification", "attenuation".
- `activation` (*Union[Callable, str], optional, default="relu"*) – Activation function to use.
- `last_activation` (*Union[Callable, str], optional, default="none"*) – Last activation function to use.
- `dropout` (*float, optional, default=0.0*) – Dropout rate.
- `residual` (*bool, optional, default=True*) – Whether to use residual connections.
- `pairwise_distances` (*bool, optional, default=False*) – Whether to use pairwise distances.
- `batch_norm` (*bool, optional, default=True*) – Whether to use batch normalization.
- `batch_norm_momentum` (*float, optional, default=0.1*) – Momentum for the batch normalization layers.
- `avg_d` (*Dict[str, float], optional, default={"log": 1.0}*) – Dictionary containing the average degree of the graph.
- `posttrans_layers` (*int, optional, default=2*) – Number of post-transformation layers.
- `pretrans_layers` (*int, optional, default=1*) – Number of pre-transformation layers.

References

Examples

```
>>> import dgl
>>> import numpy as np
>>> import torch
>>> from deepchem.models.torch_models.pna_gnn import PNALayer
>>> in_dim = 32
>>> out_dim = 64
>>> in_dim_edges = 16
>>> aggregators = ["mean", "max"]
>>> scalers = ["identity", "amplification", "attenuation"]
>>> pna_layer = PNALayer(in_dim=in_dim,
...                      out_dim=out_dim,
...                      in_dim_edges=in_dim_edges,
...                      aggregators=aggregators,
...                      scalers=scalers)
>>> num_nodes = 10
>>> num_edges = 20
>>> node_features = torch.randn(num_nodes, in_dim)
>>> edge_features = torch.randn(num_edges, in_dim_edges)
>>> g = dgl.graph((np.random.randint(0, num_nodes, num_edges),
...               np.random.randint(0, num_nodes, num_edges)))
>>> g.ndata['feat'] = node_features
>>> g.edata['feat'] = edge_features
>>> g.ndata['feat'] = pna_layer(g)
```

`__init__`(*in_dim: int*, *out_dim: int*, *in_dim_edges: int*, *aggregators: List[str]*, *scalers: List[str]*, *activation: Callable | str = 'relu'*, *dropout: float = 0.0*, *residual: bool = True*, *pairwise_distances: bool = False*, *batch_norm: bool = True*, *batch_norm_momentum=0.1*, *avg_d: Dict[str, float] = {'log': 1.0}*, *posttrans_layers: int = 2*, *pretrans_layers: int = 1*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*g*)

Forward pass of the PNA layer.

Parameters
g (*dgl.DGLGraph*) – Input graph

Returns
h – Node feature tensor

Return type
torch.Tensor

`message_func`(*edges*) → Dict[str, Tensor]

The message function to generate messages along the edges.

Parameters
edges (*dgl.EdgeBatch*) – Batch of edges.

Returns
Dictionary containing the edge features.

Return type
Dict[str, torch.Tensor]

`reduce_func`(*nodes*) → Dict[str, Tensor]

The reduce function to aggregate the messages. Apply the aggregators and scalers, and concatenate the results.

Parameters
`nodes` (*dgl.NodeBatch*) – Batch of nodes.

Returns
Dictionary containing the aggregated node features.

Return type
Dict[str, torch.Tensor]

`pretrans_edges`(*edges*) → Dict[str, Tensor]

Return a mapping to the concatenation of the features from the source node, the destination node, and the edge between them (if applicable).

Parameters
`edges` (*dgl.EdgeBatch*) – Batch of edges.

Returns
Dictionary containing the concatenated features.

Return type
Dict[str, torch.Tensor]

class `PNAGNN`(*hidden_dim*, *aggregators: List[str]*, *scalers: List[str]*, *residual: bool = True*, *pairwise_distances: bool = False*, *activation: Callable | str = 'relu'*, *batch_norm: bool = True*, *batch_norm_momentum=0.1*, *propagation_depth: int = 5*, *dropout: float = 0.0*, *posttrans_layers: int = 1*, *pretrans_layers: int = 1*, ***kwargs*)

Principal Neighbourhood Aggregation Graph Neural Network [1]. This defines the message passing layers of the PNA model.

Parameters

- `hidden_dim` (*int*) – Dimension of the hidden layers.
- `aggregators` (*List[str]*) – List of aggregator functions to use.
- `scalers` (*List[str]*) – List of scaler functions to use. Options are "identity", "amplification", "attenuation".
- `residual` (*bool, optional, default=True*) – Whether to use residual connections.
- `pairwise_distances` (*bool, optional, default=False*) – Whether to use pairwise distances.
- `activation` (*Union[Callable, str], optional, default="relu"*) – Activation function to use.
- `batch_norm` (*bool, optional, default=True*) – Whether to use batch normalization in the layers before the aggregator.
- `batch_norm_momentum` (*float, optional, default=0.1*) – Momentum for the batch normalization layers.
- `propagation_depth` (*int, optional, default=5*) – Number of propagation layers.
- `dropout` (*float, optional, default=0.0*) – Dropout rate.
- `posttrans_layers` (*int, optional, default=1*) – Number of post-transformation layers.

- **pretrans_layers** (*int, optional, default=1*) – Number of pre-transformation layers.

References

Examples

```
>>> import numpy as np
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import␣
↪RDKitConformerFeaturizer
>>> from deepchem.feat.graph_data import BatchGraphData
>>> from deepchem.models.torch_models.pna_gnn import PNAGNN
>>> featurizer = RDKitConformerFeaturizer()
>>> smiles = ['C1=CC=NC=C1', 'CC(=O)C', 'C']
>>> featurizer = RDKitConformerFeaturizer()
>>> data = featurizer.featurize(smiles)
>>> features = BatchGraphData(data)
>>> features = features.to_dgl_graph()
>>> model = PNAGNN(hidden_dim=16,
...                aggregators=['mean', 'sum'],
...                scalers=['identity'])
>>> output = model(features)
```

`__init__`(*hidden_dim, aggregators: List[str], scalers: List[str], residual: bool = True, pairwise_distances: bool = False, activation: Callable | str = 'relu', batch_norm: bool = True, batch_norm_momentum=0.1, propagation_depth: int = 5, dropout: float = 0.0, posttrans_layers: int = 1, pretrans_layers: int = 1, **kwargs*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*input_graph: DGLGraph*) → DGLGraph

Forward pass of the PNAGNN model.

Parameters

input_graph (*dgl.DGLGraph*) – Input graph with node and edge features.

Returns

graph – Output graph with updated node features after applying the message passing layers.

Return type

dgl.DGLGraph

class PNA(*hidden_dim: int, target_dim: int, task: str, aggregators: List[str] = ['mean'], scalers: List[str] = ['identity'], readout_aggregators: List[str] = ['mean'], readout_hidden_dim: int = 1, readout_layers: int = 2, residual: bool = True, pairwise_distances: bool = False, activation: Callable | str = 'relu', batch_norm: bool = True, batch_norm_momentum: float = 0.1, propagation_depth: int = 5, dropout: float = 0.0, posttrans_layers: int = 1, pretrans_layers: int = 1, n_tasks: int = 1, n_classes: int = 2, **kwargs*)

Message passing neural network for graph representation learning [1]_.

Parameters

- **hidden_dim** (*int*) – Hidden dimension size.
- **target_dim** (*int*) – Dimensionality of the output, for example for binary classification target_dim = 1.

- **aggregators** (*List[str]*) – Type of message passing functions. Options are 'mean','sum','max','min','std','var','moment3','moment4','moment5'.
- **scalers** (*List[str]*) – Type of normalization layers in the message passing network. Options are 'identity','amplification','attenuation'.
- **readout_aggregators** (*List[str]*) – Type of aggregators in the readout network.
- **readout_hidden_dim** (*int, default None*) – The dimension of the hidden layer in the readout network. If not provided, the readout has the same dimensionality of the final layer of the PNA layer, which is the hidden dimension size.
- **readout_layers** (*int, default 1*) – The number of linear layers in the readout network.
- **residual** (*bool, default True*) – Whether to use residual connections.
- **pairwise_distances** (*bool, default False*) – Whether to use pairwise distances.
- **activation** (*Union[Callable, str]*) – Activation function to use.
- **batch_norm** (*bool, default True*) – Whether to use batch normalization in the layers before the aggregator..
- **batch_norm_momentum** (*float, default 0.1*) – Momentum for the batch normalization layers.
- **propagation_depth** (*int, default*) – Number of propagation layers.
- **dropout** (*float, default 0.0*) – Dropout probability in the message passing layers.
- **posttrans_layers** (*int, default 1*) – Number of post-transformation layers.
- **pretrans_layers** (*int, default 1*) – Number of pre-transformation layers.

References

Examples

```
>>> import numpy as np
>>> from deepchem.feat.graph_data import BatchGraphData
>>> from deepchem.models.torch_models.pna_gnn import PNA
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import␣
↪RDKitConformerFeaturizer
>>> smiles = ["C1=CC=CN=C1", "C1CCC1"]
>>> featurizer = RDKitConformerFeaturizer()
>>> data = featurizer.featurize(smiles)
>>> features = BatchGraphData(data)
>>> features = features.to_dgl_graph()
>>> target_dim = 1
>>> model = PNA(hidden_dim=16, target_dim=target_dim, task='regression')
>>> output = model(features)
>>> print(output.shape)
torch.Size([1, 1])
```

__init__(*hidden_dim: int, target_dim: int, task: str, aggregators: List[str] = ['mean'], scalers: List[str] = ['identity'], readout_aggregators: List[str] = ['mean'], readout_hidden_dim: int = 1, readout_layers: int = 2, residual: bool = True, pairwise_distances: bool = False, activation: Callable | str = 'relu', batch_norm: bool = True, batch_norm_momentum: float = 0.1, propagation_depth: int = 5, dropout: float = 0.0, posttrans_layers: int = 1, pretrans_layers: int = 1, n_tasks: int = 1, n_classes: int = 2, **kwargs*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*graph: DGLGraph*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class **Net3DLayer**(*edge_dim: int*, *hidden_dim: int*, *reduce_func: str = 'sum'*, *batch_norm: bool = False*, *batch_norm_momentum: float = 0.1*, *dropout: float = 0.0*, *message_net_layers: int = 2*, *update_net_layers: int = 2*)

Net3DLayer is a single layer of a 3D graph neural network based on the 3D Infomax architecture [1].

This class expects a DGL graph with node features stored under the name 'feat' and edge features stored under the name 'd' (representing 3D distances). The edge features are updated by the message network and the node features are updated by the update network.

Parameters

- **edge_dim** (*int*) – The dimension of the edge features.
- **hidden_dim** (*int*) – The dimension of the hidden layers.
- **reduce_func** (*str*) – The reduce function to use for aggregating messages. Can be either 'sum' or 'mean'.
- **batch_norm** (*bool, optional (default=False)*) – Whether to use batch normalization.
- **batch_norm_momentum** (*float, optional (default=0.1)*) – The momentum for the batch normalization layers.
- **dropout** (*float, optional (default=0.0)*) – The dropout rate for the layers.
- **mid_activation** (*str, optional (default='SiLU')*) – The activation function to use in the network.
- **message_net_layers** (*int, optional (default=2)*) – The number of message network layers.
- **update_net_layers** (*int, optional (default=2)*) – The number of update network layers.

References

Examples

```
>>> net3d_layer = Net3DLayer(edge_dim=3, hidden_dim=3)
>>> graph = dgl.graph(([0, 1], [1, 2]))
>>> graph.ndata['feat'] = torch.tensor([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
>>> graph.edata['d'] = torch.tensor([[0.5, 0.6, 0.7], [0.8, 0.9, 1.0]])
>>> output = net3d_layer(graph)
```

`__init__`(*edge_dim: int*, *hidden_dim: int*, *reduce_func: str = 'sum'*, *batch_norm: bool = False*, *batch_norm_momentum: float = 0.1*, *dropout: float = 0.0*, *message_net_layers: int = 2*, *update_net_layers: int = 2*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*input_graph: DGLGraph*)

Perform a forward pass on the given graph.

Parameters
`input_graph` (*dgl.DGLGraph*) – The graph to perform the forward pass on.

Returns
The updated graph after the forward pass.

Return type
dgl.DGLGraph

`message_function`(*edges*)

Computes the message and edge weight for a given set of edges.

Parameters
`edges` (*dgl.EdgeBatch*) – A dgl.EdgeBatch object containing the edges information (data, batch size, etc.).

Returns
A dictionary containing the message multiplied by the edge weight.

Return type
dict

`update_function`(*nodes*)

Update function for updating node features based on the aggregated messages.

This function is used in the forward method to perform a forward pass on the graph.

Parameters
`nodes` (*dgl.NodeBatch*) – A node batch object containing the nodes information (data, batch size, etc.).

Returns
A dictionary containing the updated features.

Return type
dict

class `Net3D`(*hidden_dim*, *target_dim*, *readout_aggregators: List[str]*, *node_wise_output_layers=2*, *batch_norm=True*, *batch_norm_momentum=0.1*, *reduce_func='sum'*, *dropout=0.0*, *propagation_depth: int = 4*, *readout_layers: int = 2*, *readout_hidden_dim=None*, *fourier_encodings=4*, *update_net_layers=2*, *message_net_layers=2*, *use_node_features=False*)

Net3D is a 3D graph neural network that expects a DGL graph input with 3D coordinates stored under the name 'd' and node features stored under the name 'feat'. It is based on the 3D Infomax architecture [1].

Parameters

- `hidden_dim` (*int*) – The dimension of the hidden layers.
- `target_dim` (*int*) – The dimension of the output layer.
- `readout_aggregators` (*List[str]*) – A list of aggregator functions for the readout layer. Options are 'sum', 'max', 'min', 'mean'.

- **batch_norm** (*bool, optional (default=False)*) – Whether to use batch normalization.
- **node_wise_output_layers** (*int, optional (default=2)*) – The number of output layers for each node.
- **readout_batchnorm** (*bool, optional (default=True)*) – Whether to use batch normalization in the readout layer.
- **batch_norm_momentum** (*float, optional (default=0.1)*) – The momentum for the batch normalization layers.
- **reduce_func** (*str, optional (default='sum')*) – The reduce function to use for aggregating messages.
- **dropout** (*float, optional (default=0.0)*) – The dropout rate for the layers.
- **propagation_depth** (*int, optional (default=4)*) – The number of propagation layers in the network.
- **readout_layers** (*int, optional (default=2)*) – The number of readout layers in the network.
- **readout_hidden_dim** (*int, optional (default=None)*) – The dimension of the hidden layers in the readout network.
- **fourier_encodings** (*int, optional (default=0)*) – The number of Fourier encodings to use.
- **activation** (*str, optional (default='SiLU')*) – The activation function to use in the network.
- **update_net_layers** (*int, optional (default=2)*) – The number of update network layers.
- **message_net_layers** (*int, optional (default=2)*) – The number of message network layers.
- **use_node_features** (*bool, optional (default=False)*) – Whether to use node features as input.

Examples

```
>>> from deepchem.feat.molecule_featurizers.conformer_featurizer import␣
↪RDKitConformerFeaturizer
>>> from deepchem.models.torch_models.gnn3d import Net3D
>>> smiles = ["C[C@H](F)Cl", "C[C@@H](F)Cl"]
>>> featurizer = RDKitConformerFeaturizer()
>>> data = featurizer.featurize(smiles)
>>> dgldata = [graph.to_dgl_graph() for graph in data]
>>> net3d = Net3D(hidden_dim=3, target_dim=2, readout_aggregators=['sum', 'mean'])
>>> output = [net3d(graph) for graph in dgldata]
```

References

`__init__`(*hidden_dim*, *target_dim*, *readout_aggregators: List[str]*, *node_wise_output_layers=2*, *batch_norm=True*, *batch_norm_momentum=0.1*, *reduce_func='sum'*, *dropout=0.0*, *propagation_depth: int = 4*, *readout_layers: int = 2*, *readout_hidden_dim=None*, *fourier_encodings=4*, *update_net_layers=2*, *message_net_layers=2*, *use_node_features=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*graph: DGLGraph*)

Forward pass of the Net3D model.

Parameters
`graph` (*dgl.DGLGraph*) – The input graph with node features stored under the key 'x' and edge distances stored under the key 'd'.

Returns
The graph representation tensor of shape (1, target_dim).

Return type
torch.Tensor

`output_node_func`(*nodes*)

Apply the node-wise output network to the node features.

Parameters
`nodes` (*dgl.NodeBatch*) – A batch of nodes with features stored under the key 'feat'.

Returns
A dictionary with the updated node features under the key 'feat'.

Return type
dict

`input_edge_func`(*edges*)

Apply the edge input network to the edge features.

Parameters
`edges` (*dgl.EdgeBatch*) – A batch of edges with distances stored under the key 'd'.

Returns
A dictionary with the updated edge features under the key 'd'.

Return type
dict

`class DTNNEmbedding`(*n_embedding: int = 30*, *periodic_table_length: int = 30*, *initalizer: str = 'xavier_uniform_'*, ***kwargs*)

DTNNEmbedding layer for DTNN model.

Assign initial atomic descriptors. [1]_

This layer creates 'n' number of embeddings as initial atomic descriptors. According to the required weight initializer and periodic_table_length (Total number of unique atoms).

References

[1] Schütt, Kristof T., et al. "Quantum-chemical insights from deep
tensor neural networks." Nature communications 8.1 (2017): 1-8.

Examples

```
>>> from deepchem.models.torch_models import layers
>>> import torch
>>> layer = layers.DTNNEmbedding(30, 30, 'xavier_uniform_')
>>> output = layer(torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
>>> output.shape
torch.Size([10, 30])
```

`__init__`(*n_embedding: int = 30*, *periodic_table_length: int = 30*, *initalizer: str = 'xavier_uniform_'*, ***kwargs*)

Parameters

- **`n_embedding`** (*`int, optional`*) – Number of features for each atom
- **`periodic_table_length`** (*`int, optional`*) – Length of embedding, 83=Bi
- **`initalizer`** (*`str, optional`*) – Weight initialization for filters. Options: {**xavier_uniform_**, **xavier_normal_**, **kaiming_uniform_**, **kaiming_normal_**, **trunc_normal_**}

`forward`(*inputs: Tensor*)

Returns Embeddings according to indices.

Parameters
`inputs` (*`torch.Tensor`*) – Indices of Atoms whose embeddings are requested.

Returns
atom_embeddings – Embeddings of atoms accordings to indices.

Return type
torch.Tensor

`class DTNNStep`(*n_embedding: int = 30*, *n_distance: int = 100*, *n_hidden: int = 60*, *initializer: str = 'xavier_uniform_'*, *activation='tanh'*, ***kwargs*)

DTNNStep Layer for DTNN model.

Encodes the atom's interaction with other atoms according to distance relationships. [1]_

This Layer implements the Eq (7) from DTNN Paper. Then sums them up to get the final output using Eq (6) from DTNN Paper.

Eq (7): V_ij = tanh[W_fc . ((W_cf . C_j + b_cf) * (W_df . d_ij + b_df))]

Eq (6): C_i = C_i + sum(V_ij)

Here : '.'=Matrix Multiplication , '*'=Multiplication

References

[1] Schütt, Kristof T., et al. "Quantum-chemical insights from deep
tensor neural networks." Nature communications 8.1 (2017): 1-8.

Examples

```
>>> from deepchem.models.torch_models import layers
>>> import torch
>>> embedding_layer = layers.DTNNEmbedding(4, 4)
>>> emb = embedding_layer(torch.Tensor([0,1,2,3]).to(torch.int64))
>>> step_layer = layers.DTNNStep(4, 6, 8)
>>> output_torch = step_layer([
...     torch.Tensor(emb),
...     torch.Tensor([0, 1, 2, 3, 4, 5]).to(torch.float32),
...     torch.Tensor([1]).to(torch.int64),
...     torch.Tensor([[1]]).to(torch.int64)
... ])
>>> output_torch.shape
torch.Size([2, 4, 4])
```

`__init__`(*n_embedding: int = 30*, *n_distance: int = 100*, *n_hidden: int = 60*, *initializer: str = 'xavier_uniform_'*, *activation='tanh'*, ***kwargs*)

Parameters

- **n_embedding** (*int, optional*) – Number of features for each atom
- **n_distance** (*int, optional*) – granularity of distance matrix
- **n_hidden** (*int, optional*) – Number of nodes in hidden layer
- **initializer** (*str, optional*) – Weight initialization for filters. Options: {**xavier_uniform_**, **xavier_normal_**, **kaiming_uniform_**, **kaiming_normal_**, **trunc_normal_**}
- **activation** (*str, optional*) – Activation function applied

forward(*inputs*)

Executes the equations and Returns the intraction vector of the atom with other atoms.

Parameters
inputs (*torch.Tensor*) – List of Tensors having atom_features, distance, distance_membership_i, distance_membership_j.

Returns
interaction_vector – interaction of the atom with other atoms based on distance and distance_membership.

Return type
torch.Tensor

class DTNNGather(*n_embedding=30*, *n_outputs=100*, *layer_sizes=[100]*, *output_activation=True*, *initializer='xavier_uniform_'*, *activation='tanh'*, ***kwargs*)

DTNNGather Layer for DTNN Model.

Predict Molecular Energy using atom_features and atom_membership. [1]_

This Layer gathers the inputs got from the step layer according to atom_membership and calulates the total Molecular Energy.

References

[1] Schütt, Kristof T., et al. "Quantum-chemical insights from deep
tensor neural networks." Nature communications 8.1 (2017): 1-8.

Examples

```
>>> from deepchem.models.torch_models import layers as layers_torch
>>> import torch
>>> gather_layer_torch = layers_torch.DTNNGather(3, 3, [10])
>>> result = gather_layer_torch([torch.Tensor([[3, 2, 1]]).to(torch.float32), torch.
↪Tensor([0]).to(torch.int64)])
>>> result.shape
torch.Size([1, 3])
```

`__init__`(*n_embedding=30*, *n_outputs=100*, *layer_sizes=[100]*, *output_activation=True*, *initializer='xavier_uniform_'*, *activation='tanh'*, ***kwargs*)

Parameters

- **`n_embedding`** (*`int, optional`*) – Number of features for each atom
- **`n_outputs`** (*`int, optional`*) – Number of features for each molecule(output)
- **`layer_sizes`** (*`list of int, optional(default=[100])`*) – Structure of hidden layer(s)
- **`initializer`** (*`str, optional`*) – Weight initialization for filters.
- **`activation`** (*`str, optional`*) – Activation function applied

`forward`(*inputs*)

Executes the equation and Returns Molecular Energies according to atom_membership.

Parameters
`inputs` (*`torch.Tensor`*) – List of Tensor containing atom_features and atom_membership

Returns
molecular_energies – Tensor containing the Molecular Energies according to atom_membership.

Return type
torch.Tensor

`class GradientPenaltyLayer`(*gan:* WGANModel, *discriminator: Module*, ***kwargs*)

Implements the gradient penalty loss term for WGANs.

This class implements the gradient penalty loss term for WGANs as described in Gulrajani et al., "Improved Training of Wasserstein GANs" [wgan2]. It is used internally by WGANModel

Examples

Importing necessary modules

```
>>> import deepchem
>>> from deepchem.models.torch_models.gan import WGANModel
>>> from deepchem.models.torch_models import GradientPenaltyLayer
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
```

Creating a Generator

```
>>> class Generator(nn.Module):
...     def __init__(self, noise_input_shape, conditional_input_shape):
...         super(Generator, self).__init__()
...         self.noise_input_shape = noise_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         self.noise_dim = noise_input_shape[1:]
...         self.conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(self.noise_dim) + sum(self.conditional_dim)
...         self.output = nn.Linear(input_dim, 1)
...     def forward(self, input):
...         noise_input, conditional_input = input
...         inputs = torch.cat((noise_input, conditional_input), dim=1)
...         output = self.output(inputs)
...         return output
```

Creating a Discriminator

```
>>> class Discriminator(nn.Module):
...     def __init__(self, data_input_shape, conditional_input_shape):
...         super(Discriminator, self).__init__()
...         self.data_input_shape = data_input_shape
...         self.conditional_input_shape = conditional_input_shape
...         # Extracting the actual data dimension
...         data_dim = data_input_shape[1:]
...         # Extracting the actual conditional dimension
...         conditional_dim = conditional_input_shape[1:]
...         input_dim = sum(data_dim) + sum(conditional_dim)
...         # Define the dense layers
...         self.dense1 = nn.Linear(input_dim, 10)
...         self.dense2 = nn.Linear(10, 1)
...     def forward(self, input):
...         data_input, conditional_input = input
...         # Concatenate data_input and conditional_input along the second␣
↪dimension
...         discrim_in = torch.cat((data_input, conditional_input), dim=1)
...         # Pass the concatenated input through the dense layers
...         x = F.relu(self.dense1(discrim_in))
...         output = self.dense2(x)
...         return output
```

Creating an Example WGANModel class

```
>>> class ExampleWGAN(WGANModel):
...     def get_noise_input_shape(self):
...         return (100,2,)
...     def get_data_input_shapes(self):
...         return [(100,1,)]
...     def get_conditional_input_shapes(self):
...         return [(100,1,)]
...     def create_generator(self):
...         noise_dim = self.get_noise_input_shape()
...         conditional_dim = self.get_conditional_input_shapes()[0]
...         return nn.Sequential(Generator(noise_dim, conditional_dim))
...     def create_discriminator(self):
...         data_input_shape = self.get_data_input_shapes()[0]
...         conditional_input_shape = self.get_conditional_input_shapes()[0]
...         return nn.Sequential(
...             Discriminator(data_input_shape, conditional_input_shape))
```

Defining an Example GradientPenaltyLayer

```
>>> wgan = ExampleWGAN()
>>> discriminator = wgan.discriminators[0]
>>> gpl = GradientPenaltyLayer(wgan, discriminator)
>>> inputs = [torch.randn(4, 1)]
>>> conditional_inputs = [torch.randn(4, 1)]
>>> output, penalty = gpl(inputs, conditional_inputs)
```

References

__init__(*gan: WGANModel*, *discriminator: Module*, ***kwargs*) → None

Construct a GradientPenaltyLayer.

Parameters

- **gan** (`WGANModel`) – the WGANModel that this layer is part of
- **discriminator** (*nn.Module*) – the discriminator to compute the gradient penalty for

forward(*inputs: list | Tensor*, *conditional_inputs: Tensor*) → list

Compute the output of the gradient penalty layer.

Parameters

- **inputs** (*list of Tensor*) – the inputs to the discriminator.
- **conditional_inputs** (*Tensor*) – the conditional inputs to the discriminator.

Returns

output – the output from the discriminator, followed by the gradient penalty.

Return type

list [Tensor, Tensor]

class MolGANConvolutionLayer(*units: int*, *nodes: int*, *activation=<built-in method tanh of type object>*, *dropout_rate: float = 0.0*, *edges: int = 5*, *name: str = ''*, *prev_shape: int = 0*, *device: ~torch.device = device(type='cpu')*)

Graph convolution layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. Not used directly, higher level layers like MolGANMultiConvolutionLayer use it. This layer performs basic convolution on one-hot encoded matrices containing atom and bond information. This layer also accepts three inputs for the case when convolution is performed more than once and results of previous convolution need to used. It was done in such a way to avoid creating another layer that accepts three inputs rather than two. The last input layer is so-called hidden_layer and it hold results of the convolution while first two are unchanged input tensors.

Examples

See: MolGANMultiConvolutionLayer for using in layers.

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = 128
```

```
>>> layer1 = MolGANConvolutionLayer(units=units, edges=edges, nodes=nodes, name=
↪'layer1')
>>> adjacency_tensor = torch.randn((1, vertices, vertices, edges))
>>> node_tensor = torch.randn((1, vertices, nodes))
>>> output = layer1([adjacency_tensor, node_tensor])
```

References

`__init__`(*units: int*, *nodes: int*, *activation=<built-in method tanh of type object>*, *dropout_rate: float = 0.0*, *edges: int = 5*, *name: str = ''*, *prev_shape: int = 0*, *device: ~torch.device = device(type='cpu')*)

Initialize this layer.

Parameters

- **units** (*int*) – Dimesion of dense layers used for convolution
- **nodes** (*int*) – Number of features in node tensor
- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Dropout rate used by dropout layer
- **edges** (*int, optional (default=5)*) – How many dense layers to use in convolution. Typically equal to number of bond types used in the model.
- **name** (*string, optional (default="")*) – Name of the layer
- **prev_shape** (*int, optional (default=0)*) – Shape of the previous layer, used when more than two inputs are passed

`forward`(*inputs: List*) → Tuple[Tensor, Tensor, Tensor]

Invoke this layer

Parameters

inputs (*list*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.

Returns
First and second are original input tensors Third is the result of convolution

Return type
tuple(torch.Tensor,torch.Tensor,torch.Tensor)

`class MolGANAggregationLayer`(*units: int = 128*, *activation=<built-in method tanh of type object>*, *dropout_rate: float = 0.0*, *name: str = ''*, *prev_shape: int = 0*, *device: ~torch.device = device(type='cpu')*)

Graph Aggregation layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. Performs aggregation on tensor resulting from convolution layers. Given its simple nature it might be removed in future and moved to MolGANEncoderLayer.

Examples

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = 128
```

```
>>> layer_1 = MolGANConvolutionLayer(units=units,nodes=nodes,edges=edges, name=
↪'layer1')
>>> layer_2 = MolGANAggregationLayer(units=128, name='layer2')
>>> adjacency_tensor = torch.randn((1, vertices, vertices, edges))
>>> node_tensor = torch.randn((1, vertices, nodes))
>>> hidden_1 = layer_1([adjacency_tensor, node_tensor])
>>> output = layer_2(hidden_1[2])
```

References

`__init__`(*units: int = 128*, *activation=<built-in method tanh of type object>*, *dropout_rate: float = 0.0*, *name: str = ''*, *prev_shape: int = 0*, *device: ~torch.device = device(type='cpu')*)

Initialize the layer

Parameters

- **units** (*int, optional (default=128)*) – Dimesion of dense layers used for aggregation
- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Used by dropout layer
- **name** (*string, optional (default="")*) – Name of the layer
- **prev_shape** (*int, optional (default=0)*) – Shape of the input tensor

`forward`(*inputs: Tensor*) → Tensor

Invoke this layer

Parameters
inputs (*List*) – Single tensor resulting from graph convolution layer

Returns
aggregation tensor – Result of aggregation function on input convolution tensor.

Return type
torch.Tensor

class MolGANMultiConvolutionLayer(*units: ~typing.Tuple = (128, 64), nodes: int = 5, activation=<built-in method tanh of type object>, dropout_rate: float = 0.0, edges: int = 5, name: str = '', device: ~torch.device = device(type='cpu'), **kwargs*)

Multiple pass convolution layer used in MolGAN model. MolGAN is a WGAN type model for generation of small molecules. It takes outputs of previous convolution layer and uses them as inputs for the next one. It simplifies the overall framework, but might be moved to MolGANEncoderLayer in the future in order to reduce number of layers.

Example

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> units = (128,64)
```

```
>>> layer_1 = MolGANMultiConvolutionLayer(units=units, nodes=nodes, edges=edges,␣
↪name='layer1')
>>> adjacency_tensor = torch.randn((1, vertices, vertices, edges))
>>> node_tensor = torch.randn((1, vertices, nodes))
>>> output = layer_1([adjacency_tensor, node_tensor])
```

References

__init__(*units: ~typing.Tuple = (128, 64), nodes: int = 5, activation=<built-in method tanh of type object>, dropout_rate: float = 0.0, edges: int = 5, name: str = '', device: ~torch.device = device(type='cpu'), **kwargs*)

Initialize the layer

Parameters

- **units** (*Tuple, optional (default=(128,64)), min_length=2*) – ist of dimensions used by consecutive convolution layers. The more values the more convolution layers invoked.
- **nodes** (*int, optional (default=5)*) – Number of features in node tensor
- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Used by dropout layer
- **edges** (*int, optional (default=5)*) – Controls how many dense layers use for single convolution unit. Typically matches number of bond types used in the molecule.
- **name** (*string, optional (default="")*) – Name of the layer

forward(*inputs: List*) → Tensor

Invoke this layer

Parameters

inputs (*list*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.

Returns

convolution tensor – Result of input tensors going through convolution a number of times.

Return type

torch.Tensor

class MolGANEncoderLayer(*units: ~typing.List = [(128, 64), 128], activation: ~typing.Callable = <built-in method tanh of type object>, dropout_rate: float = 0.0, edges: int = 5, nodes: int = 5, name: str = '', device: ~torch.device = device(type='cpu'), **kwargs*)

Main learning layer used by MolGAN model. MolGAN is a WGAN type model for generation of small molecules. It role is to further simplify model. This layer can be manually built by stacking graph convolution layers followed by graph aggregation.

Example

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> vertices = 9
>>> nodes = 5
>>> edges = 5
>>> dropout_rate = 0.0
>>> adjacency_tensor = torch.randn((1, vertices, vertices, edges))
>>> node_tensor = torch.randn((1, vertices, nodes))
```

```
>>> graph = MolGANEncoderLayer(units = [(128,64),128], dropout_rate= dropout_rate,␣
↪edges=edges, nodes=nodes)([adjacency_tensor,node_tensor])
>>> dense = nn.Linear(128,128)(graph)
>>> dense = torch.tanh(dense)
>>> dense = nn.Dropout(dropout_rate)(dense)
>>> dense = nn.Linear(128,64)(dense)
>>> dense = torch.tanh(dense)
>>> dense = nn.Dropout(dropout_rate)(dense)
>>> output = nn.Linear(64,1)(dense)
```

References

__init__(*units: ~typing.List = [(128, 64), 128], activation: ~typing.Callable = <built-in method tanh of type object>, dropout_rate: float = 0.0, edges: int = 5, nodes: int = 5, name: str = '', device: ~torch.device = device(type='cpu'), **kwargs*)

Initialize the layer

Parameters

- **units** (*List, optional (default=[(128,64),128])*) – List of dimensions used by consecutive convolution layers. The more values the more convolution layers invoked.

- **activation** (*function, optional (default=Tanh)*) – activation function used across model, default is Tanh
- **dropout_rate** (*float, optional (default=0.0)*) – Used by dropout layer
- **edges** (*int, optional (default=5)*) – Controls how many dense layers use for single convolution unit. Typically matches number of bond types used in the molecule.
- **nodes** (*int, optional (default=5)*) – Number of features in node tensor
- **name** (*string, optional (default="")*) – Name of the layer

forward(*inputs: List*) → Tensor

Invoke this layer

Parameters

inputs (*list*) – List of two input matrices, adjacency tensor and node features tensors in one-hot encoding format.

Returns

encoder tensor – Tensor that been through number of convolutions followed by aggregation.

Return type

tf.Tensor

class EdgeNetwork(*n_pair_features: int = 8, n_hidden: int = 100, init: str = 'xavier_uniform_', **kwargs*)

The EdgeNetwork module is a PyTorch submodule designed for message passing in graph neural networks.

Examples

```
>>> pair_features = torch.rand((4, 2), dtype=torch.float32)
>>> atom_features = torch.rand((5, 2), dtype=torch.float32)
>>> atom_to_pair = []
>>> n_atoms = 2
>>> start = 0
>>> C0, C1 = np.meshgrid(np.arange(n_atoms), np.arange(n_atoms))
>>> atom_to_pair.append(np.transpose(np.array([C1.flatten() + start, C0.flatten() +␣
↪start])))
>>> atom_to_pair = torch.Tensor(atom_to_pair)
>>> atom_to_pair = torch.squeeze(atom_to_pair.to(torch.int64), dim=0)
>>> inputs = [pair_features, atom_features, atom_to_pair]
>>> n_pair_features = 2
>>> n_hidden = 2
>>> init = 'xavier_uniform_'
>>> layer = EdgeNetwork(n_pair_features, n_hidden, init)
>>> result = layer(inputs)
>>> result.shape[1]
2
```

__init__(*n_pair_features: int = 8, n_hidden: int = 100, init: str = 'xavier_uniform_', **kwargs*)

Initalises a EdgeNetwork Layer

Parameters

- **n_pair_features** (*int, optional*) – The length of the pair features vector.
- **n_hidden** (*int, optional*) – number of hidden units in the passing phase
- **init** (*str, optional*) – Initialization function to be used in the message passing layer.

forward(*inputs: List[Tensor]*) → Tensor

Parameters

inputs (*List[torch.Tensor]*) – The length of atom_to_pair should be same as n_pair_features.

Returns

result – Tensor containing the mapping of the edge vector to a d × d matrix, where d denotes the dimension of the internal hidden representation of each node in the graph.

Return type

torch.Tensor

class WeaveLayer(*n_atom_input_feat: int = 75*, *n_pair_input_feat: int = 14*, *n_atom_output_feat: int = 50*, *n_pair_output_feat: int = 50*, *n_hidden_AA: int = 50*, *n_hidden_PA: int = 50*, *n_hidden_AP: int = 50*, *n_hidden_PP: int = 50*, *update_pair: bool = True*, *init_: str = 'xavier_uniform_'*, *activation: str = 'relu'*, *batch_normalize: bool = True*, ***kwargs*)

This class implements the core Weave convolution from the Google graph convolution paper [1]_ This is the Torch equivalent of the original implementation using Keras.

This model contains atom features and bond features separately.Here, bond features are also called pair features. There are 2 types of transformation, atom->atom, atom->pair, pair->atom, pair->pair that this model implements.

Examples

This layer expects 4 inputs in a list of the form *[atom_features, pair_features, pair_split, atom_to_pair]*. We'll walk through the structure of these inputs. Let's start with some basic definitions.

```
>>> import deepchem as dc
>>> import numpy as np
```

Suppose you have a batch of molecules

```
>>> smiles = ["CCC", "C"]
```

Note that there are 4 atoms in total in this system. This layer expects its input molecules to be batched together.

```
>>> total_n_atoms = 4
```

Let's suppose that we have a featurizer that computes *n_atom_feat* features per atom.

```
>>> n_atom_feat = 75
```

Then conceptually, *atom_feat* is the array of shape *(total_n_atoms, n_atom_feat)* of atomic features. For simplicity, let's just go with a random such matrix.

```
>>> atom_feat = np.random.rand(total_n_atoms, n_atom_feat)
```

Let's suppose we have *n_pair_feat* pairwise features

```
>>> n_pair_feat = 14
```

For each molecule, we compute a matrix of shape *(n_atoms*n_atoms,n_pair_feat)* of pairwise features for each pair of atoms in the molecule. Let's construct this conceptually for our example.

```
>>> pair_feat = [np.random.rand(3*3, n_pair_feat), np.random.rand(1*1,n_pair_feat)]
>>> pair_feat = np.concatenate(pair_feat, axis=0)
>>> pair_feat.shape
(10, 14)
```

pair_split is an index into *pair_feat* which tells us which atom each row belongs to. In our case, we hve

```
>>> pair_split = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3])
```

That is, the first 9 entries belong to "CCC" and the last entry to "C". The final entry *atom_to_pair* goes in a little more in-depth than *pair_split* and tells us the precise pair each pair feature belongs to. In our case

```
>>> atom_to_pair = np.array([[0, 0],
...                          [0, 1],
...                          [0, 2],
...                          [1, 0],
...                          [1, 1],
...                          [1, 2],
...                          [2, 0],
...                          [2, 1],
...                          [2, 2],
...                          [3, 3]])
```

Let's now define the actual layer

```
>>> layer = WeaveLayer()
```

And invoke it

```
>>> [A, P] = layer([atom_feat, pair_feat, pair_split, atom_to_pair])
```

The weave layer produces new atom/pair features. Let's check their shapes

```
>>> A = A.detach().numpy()
>>> A.shape
(4, 50)
>>> P = P.detach().numpy()
>>> P.shape
(10, 50)
```

The 4 is *total_num_atoms* and the 10 is the total number of pairs. Where does *50* come from? It's from the default arguments *n_atom_input_feat* and *n_pair_input_feat*.

References

__init__(*n_atom_input_feat: int = 75*, *n_pair_input_feat: int = 14*, *n_atom_output_feat: int = 50*, *n_pair_output_feat: int = 50*, *n_hidden_AA: int = 50*, *n_hidden_PA: int = 50*, *n_hidden_AP: int = 50*, *n_hidden_PP: int = 50*, *update_pair: bool = True*, *init_: str = 'xavier_uniform_'*, *activation: str = 'relu'*, *batch_normalize: bool = True*, ***kwargs*)

Parameters

- **n_atom_input_feat** (*int, optional (default 75)*) – Number of features for each atom in input.

- **n_pair_input_feat** (*int, optional (default 14)*) – Number of features for each pair of atoms in input.
- **n_atom_output_feat** (*int, optional (default 50)*) – Number of features for each atom in output.
- **n_pair_output_feat** (*int, optional (default 50)*) – Number of features for each pair of atoms in output.
- **n_hidden_AA** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_hidden_PA** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_hidden_AP** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **n_hidden_PP** (*int, optional (default 50)*) – Number of units(convolution depths) in corresponding hidden layer
- **update_pair** (*bool, optional (default True)*) – Whether to calculate for pair features, could be turned off for last layer
- **init** (str, optional (default '**xavier_uniform_**')) – Weight initialization for filters.
- **activation** (*str, optional (default 'relu')*) – Activation function applied
- **batch_normalize** (*bool, optional (default True)*) – If this is turned on, apply batch normalization before applying activation functions on convolutional layers.

forward(*inputs: List[ndarray]*) → List[Tensor]

Creates weave tensors.

Parameters

inputs (*List[Union[np.ndarray, np.ndarray, np.ndarray, np.ndarray]]*) – Should contain 4 tensors [atom_features, pair_features, pair_split, atom_to_pair]

Returns

A: Atom features tensor of shape *(total_num_atoms,atom feature size)*

P: Pair features tensor of shape *(total num of pairs,bond feature size)*

Return type

List[Union[torch.Tensor, torch.Tensor]]

class WeaveGather(*batch_size: int, n_input: int = 128, gaussian_expand: bool = True, compress_post_gaussian_expansion: bool = False, init_: str = 'xavier_uniform_', activation: str = 'tanh', **kwargs*)

Implements the weave-gathering section of weave convolutions. This is the Torch equivalent of the original implementation using Keras.

Implements the gathering layer from [1]_. The weave gathering layer gathers per-atom features to create a molecule-level fingerprint in a weave convolutional network. This layer can also performs Gaussian histogram expansion as detailed in [1]_. Note that the gathering function here is simply addition as in [1]_>

Examples

This layer expects 2 inputs in a list of the form *[atom_features, pair_features]*. We'll walk through the structure of these inputs. Let's start with some basic definitions.

```
>>> import deepchem as dc
>>> import numpy as np
```

Suppose you have a batch of molecules

```
>>> smiles = ["CCC", "C"]
```

Note that there are 4 atoms in total in this system. This layer expects its input molecules to be batched together.

```
>>> total_n_atoms = 4
```

Let's suppose that we have *n_atom_feat* features per atom.

```
>>> n_atom_feat = 75
```

Then conceptually, *atom_feat* is the array of shape *(total_n_atoms, n_atom_feat)* of atomic features. For simplicity, let's just go with a random such matrix.

```
>>> atom_feat = np.random.rand(total_n_atoms, n_atom_feat)
```

We then need to provide a mapping of indices to the atoms they belong to. In ours case this would be

```
>>> atom_split = np.array([0, 0, 0, 1])
```

Let's now define the actual layer

```
>>> gather = WeaveGather(batch_size=2, n_input=n_atom_feat)
>>> output_molecules = gather([atom_feat, atom_split])
>>> len(output_molecules)
2
```

References

__init__(*batch_size: int*, *n_input: int = 128*, *gaussian_expand: bool = True*, *compress_post_gaussian_expansion: bool = False*, *init_: str = 'xavier_uniform_'*, *activation: str = 'tanh'*, ***kwargs*)

Parameters

- **batch_size** (*int*) – number of molecules in a batch
- **n_input** (*int, optional (default 128)*) – number of features for each input molecule
- **gaussian_expand** (*boolean, optional (default True)*) – Whether to expand each dimension of atomic features by gaussian histogram
- **compress_post_gaussian_expansion** (*bool, optional (default False)*) – If True, compress the results of the Gaussian expansion back to the original dimensions of the input by using a linear layer with specified activation function. Note that this compression was not in the original paper, but was present in the original DeepChem implementation so is left present for backwards compatibility.

- **init** (str, optional (default '**xavier_uniform_**')) – Weight initialization for filters if *compress_post_gaussian_expansion* is True.
- **activation** (*str, optional (default 'tanh')*) – Activation function applied for filters if *compress_post_gaussian_expansion* is True.

forward(*inputs: List[ndarray]*) → Tensor

Creates weave tensors.

Parameters

inputs (*List[Union[np.ndarray,np.ndarray]]*) – Should contain 2 tensors [atom_features, atom_split]

Returns

output_molecules – Each entry in this list is of shape *(self.n_inputs,)*

Return type

torch.Tensor

gaussian_histogram(*x: Tensor*) → Tensor

Expands input into a set of gaussian histogram bins.

Parameters

x (*torch.Tensor*) – Of shape *(N, n_feat)*

Examples

This method uses 11 bins spanning portions of a Gaussian with zero mean and unit standard deviation.

```
>>> gaussian_memberships = [(-1.645, 0.283), (-1.080, 0.170),
...                         (-0.739, 0.134), (-0.468, 0.118),
...                         (-0.228, 0.114), (0., 0.114),
...                         (0.228, 0.114), (0.468, 0.118),
...                         (0.739, 0.134), (1.080, 0.170),
...                         (1.645, 0.283)]
```

We construct a Gaussian at *gaussian_memberships[i][0]* with standard deviation *gaussian_memberships[i][1]*. Each feature in *x* is assigned the probability of falling in each Gaussian, and probabilities are normalized across the 11 different Gaussians.

Returns

outputs – Of shape *(N, 11*n_feat)*

Return type

torch.Tensor

class MXMNetGlobalMessagePassing(*dim: int*, *activation_fn: Callable | str = 'silu'*)

This class implements the Global Message Passing Layer from the Molecular Mechanics-Driven Graph Neural Network with Multiplex Graph for Molecular Structures(MXMNet) paper [1]_.

This layer consists of two message passing steps and an update step between them.

Let:

- **x_i** : `The node to be updated`
- **h_i** : `The hidden state of x_i`
- **x_j** : `The neighbour node connected to x_i by edge e_ij`
- **h_j** : `The hidden state of x_j`

- **W** : `The edge weights`
- **m_ij** : `The message between x_i and x_j`
- **h_j (self_loop)** : `The set of hidden states of atom features`
- **mlp** : `MultilayerPerceptron`
- **res** : `ResidualBlock`

In each message passing step

```
m_ij = mlp1([h_i || h_j || e_ij])*(e_ij W)
```

To handle self loops

```
m_ij = m_ij + h_j(self_loop)
```

In each update step

```
hm_j = res1(sum(m_ij))
h_j_new = mlp2(hm_j) + h_j
h_j_new = res2(h_j_new)
h_j_new = res3(h_j_new)
```

Message passing and message aggregation(sum) is handled by `propagate()`.

References

Examples

The provided example demonstrates how to use the GlobalMessagePassing layer by creating an instance, passing input tensors (node_features, edge_attributes, edge_indices) through it, and checking the shape of the output.

Initializes variables and creates a configuration dictionary with specific values.

```
>>> dim = 1
>>> node_features = torch.tensor([[0.8343], [1.2713], [1.2713], [1.2713], [1.2713]])
>>> edge_attributes = torch.tensor([[1.0004], [1.0004], [1.0005], [1.0004], [1.
↪0004],[-0.2644], [-0.2644], [-0.2644], [1.0004],[-0.2644], [-0.2644], [-0.2644],␣
↪[1.0005],[-0.2644], [-0.2644], [-0.2644], [1.0004],[-0.2644], [-0.2644], [-0.
↪2644]])
>>> edge_indices = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4,
↪ 4, 4, 4],[1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0, 1, 2, 4, 0, 1, 2, 3]])
>>> out = MXMNetGlobalMessagePassing(dim)
>>> output = out(node_features, edge_attributes, edge_indices)
>>> output.shape
torch.Size([5, 1])
```

`__init__`(*dim: int*, *activation_fn: Callable | str = 'silu'*)

Initializes the MXMNETGlobalMessagePassing layer.

Parameters

dim (*int*) – The dimension of the input and output features.

`forward`(*node_features: Tensor*, *edge_attributes: Tensor*, *edge_indices: Tensor*) → Tensor

Performs the forward pass of the GlobalMessagePassing layer.

Parameters

- **node_features** (*torch.Tensor*) – The input node features tensor of shape (num_nodes, feature_dim).
- **edge_attributes** (*torch.Tensor*) – The input edge attribute tensor of shape (num_edges, attribute_dim).
- **edge_indices** (*torch.Tensor*) – The input edge index tensor of shape (2, num_edges).

Returns

The updated node features tensor after message passing of shape (num_nodes, feature_dim).

Return type

torch.Tensor

message(*x_i: Tensor*, *x_j: Tensor*, *edge_attr: Tensor*) → Tensor

Constructs messages to be passed along the edges in the graph.

Parameters

- **x_i** (*torch.Tensor*) – The source node features tensor of shape (num_edges+num_nodes, feature_dim).
- **x_j** (*torch.Tensor*) – The target node features tensor of shape (num_edges+num_nodes, feature_dim).
- **edge_attributes** (*torch.Tensor*) – The edge attribute tensor of shape (num_edges, attribute_dim).

Returns

The constructed messages tensor.

Return type

torch.Tensor

class **MXMNetBesselBasisLayer**(*num_radial: int*, *cutoff: float = 5.0*, *envelope_exponent: int = 5*)

This layer implements a basis layer for the MXMNet model using Bessel functions. The basis layer is used to model radial symmetry in molecular systems.

The output of the layer is given by: output = envelope(dist / cutoff) * (freq * dist / cutoff).sin()

Examples

```
>>> radial_layer = MXMNetBesselBasisLayer(num_radial=2, cutoff=2.0, envelope_
↪exponent=2)
>>> distances = torch.tensor([0.5, 1.0, 2.0, 3.0])
>>> output = radial_layer(distances)
>>> output.shape
torch.Size([4, 2])
```

__init__(*num_radial: int*, *cutoff: float = 5.0*, *envelope_exponent: int = 5*)

Initialize the MXMNet Bessel Basis Layer.

Parameters

- **num_radial** (*int*) – The number of radial basis functions to use.
- **cutoff** (*float, optional (default 5.0)*) – The radial cutoff distance used to scale the distances.

- **envelope_exponent** (*int, optional (default 5)*) – The exponent of the envelope function.

reset_parameters()

Reset and initialize the learnable parameters of the MXMNet Bessel Basis Layer.

The 'freq' tensor, representing the frequencies of the Bessel functions, is set up with initial values proportional to (PI) and becomes a learnable parameter.

The 'freq' tensor will be updated during the training process to optimize the performance of the MXMNet model for the specific task it is being trained on.

forward(*dist: Tensor*) → Tensor

Compute the output of the MXMNet Bessel Basis Layer.

Parameters

dist (*torch.Tensor*) – The input tensor representing the pairwise distances between atoms.

Returns

output – The output tensor representing the radial basis functions applied to the input distances.

Return type

torch.Tensor

class DTNN(*n_tasks: int*, *n_embedding: int = 30*, *n_hidden: int = 100*, *n_distance: int = 100*, *distance_min: float = -1*, *distance_max: float = 18*, *output_activation: bool = True*, *mode: str = 'regression'*, *dropout: float = 0.0*, *n_steps: int = 2*)

Deep Tensor Neural Networks

DTNN is based on the many-body Hamiltonian concept, which is a fundamental principle in quantum mechanics. The DTNN recieves a molecule's distance matrix and membership of its atom from its Coulomb Matrix representation. Then, it iteratively refines the representation of each atom by considering its interactions with neighboring atoms. Finally, it predicts the energy of the molecule by summing up the energies of the individual atoms.

In this class, we establish a sequential model for the Deep Tensor Neural Network (DTNN) [1]_.

Examples

```
>>> import os
>>> import torch
>>> from deepchem.models.torch_models import DTNN
>>> from deepchem.data import SDFLoader
>>> from deepchem.feat import CoulombMatrix
>>> from deepchem.utils import batch_coulomb_matrix_features
>>> # Get Data
>>> model_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
>>> dataset_file = os.path.join(model_dir, 'tests/assets/qm9_mini.sdf')
>>> TASKS = ["alpha", "homo"]
>>> loader = SDFLoader(tasks=TASKS, featurizer=CoulombMatrix(29), sanitize=True)
>>> data = loader.create_dataset(dataset_file, shard_size=100)
>>> inputs = batch_coulomb_matrix_features(data.X)
>>> atom_number, distance, atom_membership, distance_membership_i, distance_
↪membership_j = inputs
>>> inputs = [torch.tensor(atom_number).to(torch.int64),
```

(continues on next page)

(continued from previous page)

```
...             torch.tensor(distance).to(torch.float32),
...             torch.tensor(atom_membership).to(torch.int64),
...             torch.tensor(distance_membership_i).to(torch.int64),
...             torch.tensor(distance_membership_j).to(torch.int64)]
>>> n_tasks = data.y.shape[0]
>>> model = DTNN(n_tasks)
>>> pred = model(inputs)
```

References

__init__(*n_tasks: int*, *n_embedding: int = 30*, *n_hidden: int = 100*, *n_distance: int = 100*, *distance_min: float = -1*, *distance_max: float = 18*, *output_activation: bool = True*, *mode: str = 'regression'*, *dropout: float = 0.0*, *n_steps: int = 2*)

Parameters

- **n_tasks** (*int*) – Number of tasks
- **n_embedding** (*int (default 30)*) – Number of features per atom.
- **n_hidden** (*int (default 100)*) – Number of features for each molecule after DTNNStep
- **n_distance** (*int (default 100)*) – granularity of distance matrix step size will be (distance_max-distance_min)/n_distance
- **distance_min** (*float (default -1)*) – minimum distance of atom pairs (in Angstrom)
- **distance_max** (*float (default 18)*) – maximum distance of atom pairs (in Angstrom)
- **output_activation** (*bool (default True)*) – determines whether an activation function should be apply to its output.
- **mode** (*str (default "regression")*) – Only "regression" is currently supported.
- **dropout** (*float (default 0.0)*) – the dropout probablity to use.
- **n_steps** (*int (default 2)*) – Number of DTNNStep Layers to use.

forward(*inputs: List[Tensor]*)

Parameters

inputs (*List*) – A list of tensors containing atom_number, distance, atom_membership, distance_membership_i, and distance_membership_j.

Returns

output – Predictions of the Molecular Energy.

Return type

torch.Tensor

class VariationalRandomizer(*embedding_dimension: int*, *annealing_start_step: int*, *annealing_final_step: int*, ***kwargs*)

Add random noise to the embedding and include a corresponding loss.

This adds random noise to the encoder, and also adds a constraint term to the loss that forces the embedding vector to have a unit Gaussian distribution. We can then pick random vectors from a Gaussian distribution, and the output sequences should follow the same distribution as the training data.

We can use this layer with an AutoEncoder, which makes it a Variational AutoEncoder. The constraint term in the loss is initially set to 0, so the optimizer just tries to minimize the reconstruction loss. Once it has made reasonable progress toward that, the constraint term can be gradually turned back on. The range of steps over which this happens is configured by modifying the annealing_start_step and annealing final_step parameter.

Examples

```
>>> from deepchem.models.torch_models.layers import VariationalRandomizer
>>> import torch
>>> embedding_dimension = 512
>>> batch_size = 100
>>> annealing_start_step = 1000
>>> annealing_final_step = 2000
>>> embedding_shape = (batch_size, embedding_dimension)
>>> embeddings = torch.rand(embedding_shape)
>>> global_step = torch.tensor([100])
>>> layer = VariationalRandomizer(embedding_dimension, annealing_start_step,␣
↪annealing_final_step)
>>> output = layer([embeddings, global_step])
>>> output.shape
torch.Size([100, 512])
```

References

__init__(*embedding_dimension: int*, *annealing_start_step: int*, *annealing_final_step: int*, ***kwargs*)

Initialize the VariationalRandomizer layer.

Parameters

- **embedding_dimension** (*int*) – The dimension of the embedding.
- **annealing_start_step** (*int*) – the step (that is, batch) at which to begin turning on the constraint term for KL cost annealing.
- **annealing_final_step** (*int*) – the step (that is, batch) at which to finish turning on the constraint term for KL cost annealing.

forward(*inputs: List[Tensor]*, *training=True*)

Returns the Variationally Randomized Embedding.

Parameters

- **inputs** (*List[torch.Tensor]*) – A list of two tensors, the first of which is the input to the layer and the second of which is the global step.
- **training** (*bool, optional (default True)*) – Whether to use the layer in training mode or inference mode.

Returns

embedding – The embedding tensor.

Return type

torch.Tensor

add_loss(*loss*)

Add a loss term to the layer.

Parameters

loss (*torch.Tensor*) – The loss tensor to add to the layer.

class EncoderRNN(*input_size: int*, *hidden_size: int*, *n_layers: int*, *dropout_p: float = 0.1*, ***kwargs*)

Encoder Layer for SeqToSeq Model.

It takes input sequences and converts them into a fixed-size context vector called the "embedding". This vector contains all relevant information from the input sequence. This context vector is then used by the decoder to generate the output sequence and can also be used as a representation of the input sequence for other Models.

Examples

```
>>> from deepchem.models.torch_models.layers import EncoderRNN
>>> import torch
>>> embedding_dimensions = 7
>>> num_input_token = 4
>>> n_layers = 9
>>> input = torch.tensor([[1, 0, 2, 3, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]])
>>> layer = EncoderRNN(num_input_token, embedding_dimensions, n_layers)
>>> emb, hidden = layer(input)
>>> emb.shape
torch.Size([3, 5, 7])
```

References

__init__(*input_size: int*, *hidden_size: int*, *n_layers: int*, *dropout_p: float = 0.1*, ***kwargs*)

Initialize the EncoderRNN layer.

Parameters

- **input_size** (*int*) – The number of expected features.
- **hidden_size** (*int*) – The number of features in the hidden state.
- **dropout_p** (*float (default 0.1)*) – The dropout probability to use during training.

forward(*input: Tensor*)

Returns Embeddings according to provided sequences.

Parameters

input (*torch.Tensor*) – Batch of input sequences.

Returns

- **output** (*torch.Tensor*) – Batch of Embeddings.
- **hidden** (*torch.Tensor*) – Batch of hidden states.

class DecoderRNN(*hidden_size: int*, *output_size: int*, *n_layers: int*, *max_length: int*, *batch_size: int*, *step_activation: str = 'relu'*, ***kwargs*)

Decoder Layer for SeqToSeq Model.

The decoder transforms the embedding vector into the output sequence. It is trained to predict the next token in the sequence given the previous tokens in the sequence. It uses the context vector from the encoder to help generate the correct token in the sequence.

Examples

```
>>> from deepchem.models.torch_models.layers import DecoderRNN
>>> import torch
>>> embedding_dimensions = 512
>>> num_output_tokens = 7
>>> max_length = 10
>>> batch_size = 100
>>> n_layers = 2
>>> layer = DecoderRNN(embedding_dimensions, num_output_tokens, n_layers, max_
↪length, batch_size)
>>> embeddings = torch.randn(batch_size, embedding_dimensions)
>>> output, hidden = layer([embeddings, None])
>>> output.shape
torch.Size([100, 10, 7])
```

References

`__init__`(*hidden_size: int*, *output_size: int*, *n_layers: int*, *max_length: int*, *batch_size: int*, *step_activation: str = 'relu'*, ***kwargs*)

Initialize the DecoderRNN layer.

Parameters

- **`hidden_size`** (*`int`*) – Number of features in the hidden state.
- **`output_size`** (*`int`*) – Number of expected features.
- **`max_length`** (*`int`*) – Maximum length of the sequence.
- **`batch_size`** (*`int`*) – Batch size of the input.
- **`step_activation`** (*`str (default "relu")`*) – Activation function to use after every step.

`forward`(*inputs: List[Tensor]*)

Parameters

`inputs` (*`List[torch.Tensor]`*) – A list of tensor containg encoder_hidden and target_tensor.

Returns

- **decoder_outputs** (*torch.Tensor*) – Predicted output sequences.
- **decoder_hidden** (*torch.Tensor*) – Hidden state of the decoder.

`class SeqToSeq`(*n_input_tokens: int*, *n_output_tokens: int*, *max_output_length: int*, *encoder_layers: int = 4*, *decoder_layers: int = 4*, *batch_size: int = 100*, *embedding_dimension: int = 512*, *dropout: float = 0.0*, *variational: bool = False*, *annealing_start_step: int = 5000*, *annealing_final_step: int = 10000*)

Implements sequence to sequence translation models.

The model is based on the description in Sutskever et al., "Sequence to Sequence Learning with Neural Networks" (https://arxiv.org/abs/1409.3215), although this implementation uses GRUs instead of LSTMs. The goal is to take sequences of tokens as input, and translate each one into a different output sequence. The input and output sequences can both be of variable length, and an output sequence need not have the same length as the input sequence it was generated from. For example, these models were originally developed for use in natural language processing. In that context, the input might be a sequence of English words, and the output might be a sequence of French words. The goal would be to train the model to translate sentences from English to French.

The model consists of two parts called the "encoder" and "decoder". Each one consists of a stack of recurrent layers. The job of the encoder is to transform the input sequence into a single, fixed length vector called the "embedding". That vector contains all relevant information from the input sequence. The decoder then transforms the embedding vector into the output sequence.

These models can be used for various purposes. First and most obviously, they can be used for sequence to sequence translation. In any case where you have sequences of tokens, and you want to translate each one into a different sequence, a SeqToSeq model can be trained to perform the translation.

Another possible use case is transforming variable length sequences into fixed length vectors. Many types of models require their inputs to have a fixed shape, which makes it difficult to use them with variable sized inputs (for example, when the input is a molecule, and different molecules have different numbers of atoms). In that case, you can train a SeqToSeq model as an autoencoder, so that it tries to make the output sequence identical to the input one. That forces the embedding vector to contain all information from the original sequence. You can then use the encoder for transforming sequences into fixed length embedding vectors, suitable to use as inputs to other types of models.

Another use case is to train the decoder for use as a generative model. Here again you begin by training the SeqToSeq model as an autoencoder. Once training is complete, you can supply arbitrary embedding vectors, and transform each one into an output sequence. When used in this way, you typically train it as a variational autoencoder. This adds random noise to the encoder, and also adds a constraint term to the loss that forces the embedding vector to have a unit Gaussian distribution. You can then pick random vectors from a Gaussian distribution, and the output sequences should follow the same distribution as the training data.

When training as a variational autoencoder, it is best to use KL cost annealing, as described in https://arxiv.org/abs/1511.06349. The constraint term in the loss is initially set to 0, so the optimizer just tries to minimize the reconstruction loss. Once it has made reasonable progress toward that, the constraint term can be gradually turned back on. The range of steps over which this happens is configurable.

In this class, we establish a sequential model for the Sequence to Sequence (SeqToSeq) **[1]_**.

Examples

```
>>> import torch
>>> from deepchem.models.torch_models.seqtoseq import SeqToSeq
>>> from deepchem.utils.batch_utils import create_input_array
>>> # Dataset of SMILES strings for testing SeqToSeq models.
>>> train_smiles = [
...     'Cc1cccc(N2CCN(C(=O)C34CC5CC(CC(C5)C3)C4)CC2)c1C',
...     'Cn1ccnc1SCC(=O)Nc1ccc(Oc2ccccc2)cc1',
...     'COc1cc2c(cc1NC(=O)CN1C(=O)NC3(CCc4ccccc43)C1=O)oc1ccccc12',
...     'CCCc1cc(=O)nc(SCC(=O)N(CC(C)C)C2CCS(=O)(=O)C2)[nH]1',
... ]
>>> tokens = set()
>>> for s in train_smiles:
...     tokens = tokens.union(set(c for c in s))
>>> token_list = sorted(list(tokens))
>>> batch_size = len(train_smiles)
>>> MAX_LENGTH = max(len(s) for s in train_smiles)
>>> token_list = token_list + [" "]
>>> input_dict = dict((x, i) for i, x in enumerate(token_list))
>>> n_tokens = len(token_list)
```

(continues on next page)

(continued from previous page)

```
>>> embedding_dimension = 16
>>> model = SeqToSeq(n_tokens, n_tokens, MAX_LENGTH, batch_size=batch_size,
...                  embedding_dimension=embedding_dimension)
>>> inputs = create_input_array(train_smiles, MAX_LENGTH, False, batch_size,
...                             input_dict, " ")
>>> output, embeddings = model([torch.tensor(inputs), torch.tensor([1])])
>>> output.shape
torch.Size([4, 57, 19])
>>> embeddings.shape
torch.Size([4, 16])
```

References

`__init__`(*n_input_tokens: int*, *n_output_tokens: int*, *max_output_length: int*, *encoder_layers: int = 4*, *decoder_layers: int = 4*, *batch_size: int = 100*, *embedding_dimension: int = 512*, *dropout: float = 0.0*, *variational: bool = False*, *annealing_start_step: int = 5000*, *annealing_final_step: int = 10000*)

Initialize SeqToSeq model.

Parameters

- **`n_input_tokens`** (*`int`*) – Number of input tokens.
- **`n_output_tokens`** (*`int`*) – Number of output tokens.
- **`max_output_length`** (*`int`*) – Maximum length of output sequence.
- **`encoder_layers`** (*`int (default 4)`*) – Number of recurrent layers in the encoder
- **`decoder_layers`** (*`int (default 4)`*) – Number of recurrent layers in the decoder
- **`embedding_dimension`** (*`int (default 512)`*) – Width of the embedding vector. This also is the width of all recurrent layers.
- **`dropout`** (*`float (default 0.0)`*) – Dropout probability to use during training.
- **`variational`** (*`bool (default False)`*) – If True, train the model as a variational autoencoder. This adds random noise to the encoder, and also constrains the embedding to follow a unit Gaussian distribution.
- **`annealing_start_step`** (*`int (default 5000)`*) – the step (that is, batch) at which to begin turning on the constraint term for KL cost annealing.
- **`annealing_final_step`** (*`int (default 10000)`*) – the tep (that is, batch) at which to finish turning on the constraint term for KL cost annealing.

`forward`(*inputs: List*)

Generates Embeddings using Encoder then passes it to Decoder to predict output sequences.

Parameters

`inputs` (*`List`*) – List of two tensors. First tensor is batch of input sequence. Second tensor is the current global_step.

Returns

- **output** (*torch.Tensor*) – Predicted output sequence.
- **_embedding** (*torch.Tensor*) – Embeddings generated by the Encoder.

class **FerminetElectronFeature**(*n_one: List[int]*, *n_two: List[int]*, *no_of_atoms: int*, *batch_size: int*, *total_electron: int*, *spin: List[int]*)

A Pytorch Module implementing the ferminet's electron features interaction layer _[1]. This is a helper class for the Ferminet model.

The layer consists of 2 types of linear layers - v for the one elctron features and w for the two electron features. The number and dimensions of each layer depends on the number of atoms and electrons in the molecule system.

References

Examples

```
>>> import deepchem as dc
>>> electron_layer = dc.models.torch_models.layers.FerminetElectronFeature([32,32,
↪32],[16,16,16], 4, 8, 10, [5,5])
>>> one_electron_test = torch.randn(8, 10, 4*4)
>>> two_electron_test = torch.randn(8, 10, 10, 4)
>>> one, two = electron_layer.forward(one_electron_test, two_electron_test)
>>> one.size()
torch.Size([8, 10, 32])
>>> two.size()
torch.Size([8, 10, 10, 16])
```

__init__(*n_one: List[int]*, *n_two: List[int]*, *no_of_atoms: int*, *batch_size: int*, *total_electron: int*, *spin: List[int]*)

Parameters

- **n_one** (*List[int]*) – List of integer values containing the dimensions of each n_one layer's output
- **n_two** (*List[int]*) – List of integer values containing the dimensions of each n_one layer's output
- **no_of_atoms** (*int:*) – Value containing the number of atoms in the molecule system
- **batch_size** (*int*) – Value containing the number of batches for the input provided
- **total_electron** (*int*) – Value containing the total number of electrons in the molecule system
- **spin** (*List[int]*) – List data structure in the format of [number of up-spin electrons, number of down-spin electrons]
- **v** (*torch.nn.ModuleList*) – torch ModuleList containing the linear layer with the n_one layer's dimension size.
- **w** (*torch.nn.ModuleList*) – torch ModuleList containing the linear layer with the n_two layer's dimension size.
- **layer_size** (*int*) – Value containing the number of n_one and n_two layers

forward(*one_electron: Tensor*, *two_electron: Tensor*)

Parameters

- **`one_electron`** (*`torch.Tensor`*) – The one electron feature which has the shape (batch_size, number of electrons, number of atoms * 4). Here the last dimension contains the electron's distance from each of the atom as a vector concatenated with norm of that vector.
- **`two_electron`** (*`torch.Tensor`*) – The two electron feature which has the shape (batch_size, number of electrons, number of electron , 4). Here the last dimension contains the electron's distance from the other electrons as a vector concatenated with norm of that vector.

Returns

- **one_electron** (*torch.Tensor*) – The one electron feature after passing through the layer which has the shape (batch_size, number of electrons, n_one shape).
- **two_electron** (*torch.Tensor*) – The two electron feature after passing through the layer which has the shape (batch_size, number of electrons, number of electron , n_two shape).

`class` **`FerminetEnvelope`**(*n_one: List[int]*, *n_two: List[int]*, *total_electron: int*, *batch_size: int*, *spin: List[int]*, *no_of_atoms: int*, *determinant: int*)

A Pytorch Module implementing the ferminet's envlope layer _[1], which is used to calculate the spin up and spin down orbital values. This is a helper class for the Ferminet model. The layer consists of 4 types of parameter lists - envelope_w, envelope_g, sigma and pi, which helps to calculate the orbital vlaues.

References

Examples

```
>>> import deepchem as dc
>>> import torch
>>> envelope_layer = dc.models.torch_models.layers.FerminetEnvelope([32, 32, 32],␣
↪[16, 16, 16], 10, 8, [5, 5], 5, 16)
>>> one_electron = torch.randn(8, 10, 32)
>>> one_electron_permuted = torch.randn(8, 10, 5, 3)
>>> psi, psi_up, psi_down = envelope_layer.forward(one_electron, one_electron_
↪permuted)
>>> psi.size()
torch.Size([8])
>>> psi_up.size()
torch.Size([8, 16, 5, 5])
>>> psi_down.size()
torch.Size([8, 16, 5, 5])
```

`__init__`(*n_one: List[int]*, *n_two: List[int]*, *total_electron: int*, *batch_size: int*, *spin: List[int]*, *no_of_atoms: int*, *determinant: int*)

Parameters

- **`n_one`** (*`List[int]`*) – List of integer values containing the dimensions of each n_one layer's output
- **`n_two`** (*`List[int]`*) – List of integer values containing the dimensions of each n_one layer's output
- **`total_electron`** (*`int`*) – Value containing the total number of electrons in the molecule system

- **batch_size** (*int*) – Value containing the number of batches for the input provided
- **spin** (*List[int]*) – List data structure in the format of [number of up-spin electrons, number of down-spin electrons]
- **no_of_atoms** (*int*) – Value containing the number of atoms in the molecule system
- **determinant** (*int*) – The number of determinants to be incorporated in the post-HF solution.
- **envelope_w** (*torch.nn.ParameterList*) – torch ParameterList containing the torch Tensor with n_one layer's dimension size.
- **envelope_g** (*torch.nn.ParameterList*) – torch ParameterList containing the torch Tensor with the unit dimension size, which acts as bias.
- **sigma** (*torch.nn.ParameterList*) – torch ParameterList containing the torch Tensor with the unit dimension size.
- **pi** (*torch.nn.ParameterList*) – torch ParameterList containing the linear layer with the n_two layer's dimension size.
- **layer_size** (*int*) – Value containing the number of n_one and n_two layers

forward(*one_electron: Tensor*, *one_electron_vector_permuted: Tensor*)

Parameters

- **one_electron** (*torch.Tensor*) – Torch tensor which is output from FerminElectronFeature layer in the shape of (batch_size, number of elctrons, n_one layer size).
- **one_electron_vector_permuted** (*torch.Tensor*) – Torch tensor which is shape permuted vector of the original one_electron vector tensor. shape of the tensor should be (batch_size, number of atoms, number of electrons, 3).

Returns

psi_up – Torch tensor with a scalar value containing the sampled wavefunction value for each batch.

Return type

torch.Tensor

class MXMNetLocalMessagePassing(*dim: int*, *activation_fn: Callable | str = 'silu'*)

The MXMNetLocalMessagePassing class defines a local message passing layer used in the MXMNet model [1]_. This layer integrates cross-layer mappings inside the local message passing, allowing for the transformation of input tensors representing pairwise distances and angles between atoms in a molecular system. The layer aggregates information using message passing and updates atom representations accordingly. The 3-step message passing scheme is proposed in the paper [1]_.

1. Step 1 contains Message Passing 1 that captures the two-hop angles and related pairwise distances to update edge-level embeddings {mji}.
2. Step 2 contains Message Passing 2 that captures the one-hop angles and related pairwise distances to further update {mji}.
3. Step 3 finally aggregates {mji} to update the node-level embedding hi.

These steps in the t-th iteration can be formulated as follows:

Let:

- **mlp** : `MultilayerPerceptron`
- **res** : `ResidualBlock`

- **h** : node_features
- **m** : message with radial basis function
- **idx_kj**: Tensor containing indices for the k and j atoms
- **x_i** : The node to be updated
- **h_i** : The hidden state of x_i
- **x_j** : The neighbour node connected to x_i by edge e_ij
- **h_j** : The hidden state of x_j
- **rbf** : Input tensor representing radial basis functions
- **sbf** : Input tensor representing the spherical basis functions
- **idx_jj** : Tensor containing indices for the j and j' where j' is other neighbours of i

Step 1: Message Passing 1

```
m = [h[i] || h[j] || rbf]
m_kj = mlp_kj(m[idx_kj]) * (rbf*W) * mlp_sbf1(sbf1)
m_ji = mlp_ji_1(m) + reduce_sum(m_kj)
```

Step 2: Message Passing 2

```
m_ji = mlp_jj(m_ji[idx_jj]) * (rbf*W) * mlp_sbf2(sbf2)
m_ji = mlp_ji_2(m_ji) + reduce_sum(m_ji)
```

Step 3: Aggregation and Update

In each aggregation step

```
m = reduce_sum(m_ji*(rbf*W))
```

In each update step

```
hm_i = res1(m)
h_i_new = mlp2(hm_i) + h_i
h_i_new = res2(h_i_new)
h_i_new = res3(h_i_new)
```

References

Examples

```
>>> dim = 1
>>> h = torch.tensor([[0.8343], [1.2713], [1.2713], [1.2713], [1.2713]])
>>> rbf = torch.tensor([[-0.2628], [-0.2628], [-0.2628], [-0.2628],
...                      [-0.2629], [-0.2629], [-0.2628], [-0.2628]])
>>> sbf1 = torch.tensor([[-0.2767], [-0.2767], [-0.2767], [-0.2767],
...                       [-0.2767], [-0.2767], [-0.2767], [-0.2767],
...                       [-0.2767], [-0.2767], [-0.2767], [-0.2767]])
>>> sbf2 = torch.tensor([[-0.0301], [-0.0301], [-0.1483], [-0.1486], [-0.1484],
...                       [-0.0301], [-0.1483], [-0.0301], [-0.1485], [-0.1483],
```

(continues on next page)

(continued from previous page)

```
...                         [-0.0301], [-0.1486], [-0.1485], [-0.0301], [-0.1486],
...                         [-0.0301], [-0.1484], [-0.1483], [-0.1486], [-0.0301]])
>>> idx_kj = torch.tensor([3, 5, 7, 1, 5, 7, 1, 3, 7, 1, 3, 5])
>>> idx_ji_1 = torch.tensor([0, 0, 0, 2, 2, 2, 4, 4, 4, 6, 6, 6])
>>> idx_jj = torch.tensor([0, 1, 3, 5, 7, 2, 1, 3, 5, 7, 4, 1, 3, 5, 7, 6, 1, 3, 5,␣
↪7])
>>> idx_ji_2 = torch.tensor([0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 4, 5, 5, 5, 5, 6, 7, 7,␣
↪7, 7])
>>> edge_index = torch.tensor([[0, 1, 0, 2, 0, 3, 0, 4],
...                            [1, 0, 2, 0, 3, 0, 4, 0]])
>>> out = MXMNetLocalMessagePassing(dim, activation_fn='silu')
>>> output = out(h,
...             rbf,
...             sbf1,
...             sbf2,
...             idx_kj,
...             idx_ji_1,
...             idx_jj,
...             idx_ji_2,
...             edge_index)
>>> output[0].shape
torch.Size([5, 1])
>>> output[1].shape
torch.Size([5, 1])
```

__init__(*dim: int*, *activation_fn: Callable | str = 'silu'*)

Initializes the MXMNetLocalMessagePassing layer.

Parameters

- **dim** (*int*) – The dimension of the input and output tensors for the local message passing layer.
- **activation_fn** (*Union[Callable, str], optional (default: 'silu')*) – The activation function to be used in the multilayer perceptrons (MLPs) within the layer.

forward(*node_features: Tensor*, *rbf: Tensor*, *sbf1: Tensor*, *sbf2: Tensor*, *idx_kj: Tensor*, *idx_ji_1: Tensor*, *idx_jj: Tensor*, *idx_ji_2: Tensor*, *edge_index: Tensor*) → Tuple[Tensor, Tensor]

The forward method performs the computation for the MXMNetLocalMessagePassing Layer. This method processes the input tensors representing atom features, radial basis functions (RBF), and spherical basis functions (SBF) using message passing over the molecular graph. The message passing updates the atom representations, and the resulting tensor represents the updated atom feature after local message passing.

Parameters

- **node_features** (*torch.Tensor*) – Input tensor representing atom features.
- **rbf** (*torch.Tensor*) – Input tensor representing radial basis functions.
- **sbf1** (*torch.Tensor*) – Input tensor representing the first set of spherical basis functions.
- **sbf2** (*torch.Tensor*) – Input tensor representing the second set of spherical basis functions.
- **idx_kj** (*torch.Tensor*) – Tensor containing indices for the k and j atoms involved in each interaction.

- **idx_ji_1** (*torch.Tensor*) – Tensor containing indices for the j and i atoms involved in the first message passing step.
- **idx_jj** (*torch.Tensor*) – Tensor containing indices for the j and j' atoms involved in the second message passing step.
- **idx_ji_2** (*torch.Tensor*) – Tensor containing indices for the j and i atoms involved in the second message passing step.
- **edge_index** (*torch.Tensor*) – Tensor containing the edge indices of the molecular graph, with shape (2, M), where M is the number of edges.

Returns

- **node_features** (*torch.Tensor*) – Updated atom representations after local message passing.
- **output** (*torch.Tensor*) – Output tensor representing a fixed-size representation, with shape (N, 1).

class **MXMNetSphericalBasisLayer**(*num_spherical: int*, *num_radial: int*, *cutoff: float = 5.0*, *envelope_exponent: int = 5*)

It takes pairwise distances and angles between atoms as input and combines radial basis functions with spherical harmonic functions to generate a fixed-size representation that captures both radial and orientation information. This type of representation is commonly used in molecular modeling and simulations to capture the behavior of atoms and molecules in chemical systems.

Inside the initialization, Bessel basis functions and real spherical harmonic functions are generated. The Bessel basis functions capture the radial information, and the spherical harmonic functions capture the orientation information. These functions are generated based on the provided num_spherical and num_radial parameters.

Examples

```
>>> dist = torch.tensor([0.5, 1.0, 2.0, 3.0])
>>> angle = torch.tensor([0.1, 0.2, 0.3, 0.4])
>>> idx_kj = torch.tensor([0, 1, 2, 3])
>>> spherical_layer = MXMNetSphericalBasisLayer(envelope_exponent=2, num_
↪spherical=2, num_radial=2, cutoff=2.0)
>>> output = spherical_layer(dist, angle, idx_kj)
>>> output.shape
torch.Size([4, 4])
```

__init__(*num_spherical: int*, *num_radial: int*, *cutoff: float = 5.0*, *envelope_exponent: int = 5*)

Initialize the MXMNetSphericalBasisLayer.

Parameters

- **num_spherical** (*int*) – The number of spherical harmonic functions to use. These functions capture orientation information related to atom positions.
- **num_radial** (*int*) – The number of radial basis functions to use. These functions capture information about pairwise distances between atoms.
- **cutoff** (*float, optional (default 5.0)*) – The cutoff distance for the radial basis functions. It specifies the distance beyond which the interactions are ignored.
- **envelope_exponent** (*int, optional (default 5)*) – The exponent for the envelope function. It controls the degree of damping for the radial basis functions.

forward(*dist: Tensor*, *angle: Tensor*, *idx_kj: Tensor*) → Tensor

Forward pass of the MXMNetSphericalBasisLayer.

Parameters

- **dist** (*torch.Tensor*) – Input tensor representing pairwise distances between atoms.
- **angle** (*torch.Tensor*) – Input tensor representing pairwise angles between atoms.
- **idx_kj** (*torch.Tensor*) – Tensor containing indices for the k and j atoms.

Returns

output – The output tensor containing the fixed-size representation.

Return type

torch.Tensor

class HighwayLayer(*d_input: int*, *activation_fn: Callable | str = 'relu'*)

Highway layer from "Training Very Deep Networks" [1]

y = H(x) * T(x) + x * C(x), where

H(x): 1-layer neural network with non-linear activation T(x): 1-layer neural network with sigmoid activation C(X): 1 - T(X); As per the original paper

The output will be of the same dimension as the input

References

Examples

```
>>> x = torch.randn(16, 20)
>>> highway_layer = HighwayLayer(d_input=x.shape[1])
>>> y = highway_layer(x)
>>> x.shape
torch.Size([16, 20])
>>> y.shape
torch.Size([16, 20])
```

__init__(*d_input: int*, *activation_fn: Callable | str = 'relu'*)

Initializes the HighwayLayer.

Parameters

- **d_input** (*int*) – the dimension of the input layer
- **activation_fn** (*str*) – the activation function to use for H(x)

forward(*x: Tensor*) → Tensor

Forward pass of the HighwayLayer.

Parameters

x (*torch.Tensor*) – Input tensor of dimension (,input_dim).

Returns

output – Output tensor of dimension (,input_dim)

Return type

torch.Tensor

class ClampExp(*lambda_param: float = 1.0*)

A non Linearity layer that clamps the input tensor by taking the minimum of the exponential of the input multiplied by a lambda parameter and 1.

$$f(x) = min(exp(\lambda * x), 1)$$

Example

```
>>> import torch
>>> from deepchem.models.torch_models.flows import ClampExp
>>> lambda_param = 1.0
>>> clamp_exp = ClampExp(lambda_param)
>>> input = torch.tensor([-1 ,0.5, 0.6, 0.7])
>>> clamp_exp(input)
tensor([0.3679, 1.0000, 1.0000, 1.0000])
```

__init__(*lambda_param: float = 1.0*) → None

Initializes the ClampExp layer

Parameters

lambda_param (*float*) – Lambda parameter for the ClampExp layer

forward(*x: Tensor*) → Tensor

Forward pass of the ClampExp layer

Parameters

x (*torch.Tensor*) – Input tensor

Returns

Transformed tensor according to ClampExp layer with the shape of 'x'.

Return type

torch.Tensor

class ConstScaleLayer(*scale: float = 1.0*)

This layer scales the input tensor by a fixed factor

Example

```
>>> import torch
>>> from deepchem.models.torch_models.flows import ConstScaleLayer
>>> scale = 2.0
>>> const_scale = ConstScaleLayer(scale)
>>> input = torch.tensor([1, 2, 3])
>>> const_scale(input)
tensor([2., 4., 6.])
```

__init__(*scale: float = 1.0*)

Initializes the ConstScaleLayer

Parameters

scale (*float*) – Scaling factor

forward(*input: Tensor*) → Tensor

Forward pass of the ConstScaleLayer

Parameters
`input` (*torch.Tensor*) – Input tensor

Returns
Scaled tensor

Return type
torch.Tensor

Flow Layers

class Flow

Generic class for flow functions

Flows [flow1] should satisfy several conditions in order to be practical. They should:

- be invertible; for sampling we need g while for computing likelihood we need f ,
- be sufficiently expressive to model the distribution of interest,
- be computationally efficient, both in terms of computing f and g (depending on the application) but also in terms of the calculation of the determinant of the Jacobian.

Flow layers are generally used as a part of a Normalizing Flow model, which is a generative model that learns a target distribution by transforming a simple base distribution through a series of invertible transformations. The target distribution is then defined as the composition of the base distribution and the flow transformations.

References

__init__()

Initializes the flow function

forward(*z: Tensor*) → Tuple[Tensor, Tensor]

Forward pass of the flow

Parameters
`z` (*torch.Tensor*) – Input tensor

Returns

- **z_** (*torch.Tensor*) – Transformed tensor
- **log_det** (*torch.Tensor*) – Logarithm of the determinant of the Jacobian of the transformation

inverse(*z: Tensor*) → Tuple[Tensor, Tensor]

Inverse pass of the flow

Parameters
`z` (*torch.Tensor*) – Input tensor

Returns

- **z_** (*torch.Tensor*) – Transformed tensor
- **log_det** (*torch.Tensor*) – Logarithm of the determinant of the Jacobian of the transformation

class Affine(*dim: int*)

Class which performs the Affine transformation.

This transformation is based on the affinity of the base distribution with the target distribution. A geometric transformation is applied where the parameters performs changes on the scale and shift of a function (inputs).

Normalizing Flow transformations must be bijective in order to compute the logarithm of jacobian's determinant. For this reason, transformations must perform a forward and inverse pass.

Example

```
>>> import deepchem as dc
>>> from deepchem.models.torch_models.layers import Affine
>>> import torch
>>> from torch.distributions import MultivariateNormal
>>> # initialize the transformation layer's parameters
>>> dim = 2
>>> samples = 96
>>> transforms = Affine(dim)
>>> # forward pass based on a given distribution
>>> distribution = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> input = distribution.sample(torch.Size((samples, dim)))
>>> len(transforms.forward(input))
2
>>> # inverse pass based on a distribution
>>> len(transforms.inverse(input))
2
```

__init__(*dim: int*) → None

Create a Affine transform layer.

Parameters

dim (*int*) – Value of the Nth dimension of the dataset.

forward(*x: Tensor*) → Tuple[Tensor, Tensor]

Performs a transformation between two different distributions. This particular transformation represents the following function:

$$y = x * exp(a) + b$$

where a is scale parameter and b performs a shift. This class also returns the logarithm of the jacobians determinant which is useful when invert a transformation and compute the probability of the transformation.

Parameters

x (*torch.Tensor*) – Tensor sample with the initial distribution data which will pass into the normalizing flow algorithm.

Returns

- **y** (*torch.Tensor*) – Transformed tensor according to Affine layer with the shape of 'x'.
- **log_det_jacobian** (*torch.Tensor*) – Tensor which represents the info about the deviation of the initial and target distribution.

inverse(*y: Tensor*) → Tuple[Tensor, Tensor]

Performs a transformation between two different distributions. This transformation represents the bacward pass of the function mention before. Its mathematical representation is x = (y - b) / exp(a) , where "a" is scale parameter and "b" performs a shift. This class also returns the logarithm of the jacobians determinant which is useful when invert a transformation and compute the probability of the transformation.

Parameters

y (*torch.Tensor*) – Tensor sample with transformed distribution data which will be used in the normalizing algorithm inverse pass.

Returns

- **x** (*torch.Tensor*) – Transformed tensor according to Affine layer with the shape of 'y'.
- **inverse_log_det_jacobian** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

class MaskedAffineFlow(*b: Tensor*, *t: ModuleList | Sequential | None = None*, *s: ModuleList | Sequential | None = None*)

This class implements the Masked Affine Flow layer

The Masked Affine Flow [maskedaffine1] layer is a type of normalizing flow layer which is used to learn a target distribution. The layer is based on the affine flow layer, but with a mask applied to the input data. The mask is a tensor of the same size as the input data, filled with 0s and 1s. The mask is used to determine which features are transformed by the affine flow layer. The affine flow layer is defined as follows:

Masked affine flow .. math:: f(z) = b * z + (1 - b) * (z * e^{s(b * z)} + t)

Example

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> from deepchem.models.torch_models.flows import MaskedAffineFlow
>>> from torch.distributions import MultivariateNormal
```

```
>>> dim = 2
>>> samples = 96
>>> data = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> tensor = data.sample(torch.Size((samples, dim)))
```

```
>>> layers = 4
>>> hidden_size = 16
>>> masks = F.one_hot(torch.tensor([i % 2 for i in range(layers)])).float()
```

```
>>> s_func = nn.Sequential(
...     nn.Linear(in_features=dim, out_features=hidden_size), nn.LeakyReLU(),
...     nn.Linear(in_features=hidden_size, out_features=hidden_size),
...     nn.LeakyReLU(), nn.Linear(in_features=hidden_size, out_features=dim))
```

```
>>> t_func = nn.Sequential(
...     nn.Linear(in_features=dim, out_features=hidden_size), nn.LeakyReLU(),
...     nn.Linear(in_features=hidden_size, out_features=hidden_size),
...     nn.LeakyReLU(), nn.Linear(in_features=hidden_size, out_features=dim))
```

```
>>> layers = nn.ModuleList(
...     [MaskedAffineFlow(mask, s_func, t_func) for mask in masks])
```

```
>>> for layer in layers:
...   _, inverse_log_det_jacobian = layer.inverse(tensor)
...   inverse_log_det_jacobian = inverse_log_det_jacobian.detach().numpy()
>>> len(inverse_log_det_jacobian)
96
```

References

`__init__`(*b: Tensor*, *t: ModuleList | Sequential | None = None*, *s: ModuleList | Sequential | None = None*) → None

Initializes the Masked Affine Flow layer

Parameters

- **b** (*torch.Tensor*) – mask for features, i.e. tensor of same size as latent data point filled with 0s and 1s
- **t** (*Optional[Union[torch.nn.ModuleList, torch.nn.Sequential]], optional*) – translation mapping, i.e. neural network, where first input dimension is batch dim, if None no translation is applied
- **s** (*Optional[Union[torch.nn.ModuleList, torch.nn.Sequential]], optional*) – scale mapping, i.e. neural network, where first input dimension is batch dim, if None no scale is applied

`forward`(*z: Tensor*) → Tuple[Tensor, Tensor]

Forward pass of the Masked Affine Flow layer

Parameters

z (*torch.Tensor*) – Input tensor

Returns

- **z** (*torch.Tensor*) – Transformed tensor according to Masked Affine Flow layer with the shape of 'z'.
- **log_det** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

`inverse`(*z: Tensor*) → Tuple[Tensor, Tensor]

Inverse pass of the Masked Affine Flow layer

Parameters

z (*torch.Tensor*) – Input tensor

Returns

- **z_** (*torch.Tensor*) – Transformed tensor according to Masked Affine Flow layer with the shape of 'z'.
- **log_det** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

class `ActNorm`(**args*, ***kwargs*)

This class implements the ActNorm layer (for activation normalizaton)

ActNorm is an Affine layer but with a data-dependent initialization, where on the very first batch we clever initialize the scale,shift so that the output is unit gaussian. As described in [glow] Kingma et al (2018).

ActNorm is a layer that performs an affine transformation of the activations using a scale and bias parameter per channel, similar to batch normalization. These parameters are initialized such that the post-actnorm activations per-channel have zero mean and unit variance given an initial minibatch of data. This is a form of data dependent initialization [weight_norm]. After initialization, the scale and bias are treated as regular trainable parameters that are independent of the data.

Examples

Importing necessary libraries

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> from deepchem.models.torch_models.flows import MaskedAffineFlow
>>> from torch.distributions import MultivariateNormal
```

Creating sample data

```
>>> dim = 2
>>> samples = 96
>>> data = MultivariateNormal(torch.zeros(dim), torch.eye(dim))
>>> tensor = data.sample(torch.Size((samples, dim)))
```

Initializing the ActNorm layer and performing forward and inverse pass

```
>>> actnorm = ActNorm(dim)
>>> _, log_det_jacobian = actnorm.forward(tensor)
>>> _, inverse_log_det_jacobian = actnorm.inverse(tensor)
>>> len(inverse_log_det_jacobian)
96
```

References

`__init__`(**args*, ***kwargs*) → None

Initializes the ActNorm layer

`forward`(*z: Tensor*) → Tuple[Tensor, Tensor]

Forward pass of the ActNorm layer

Parameters
z (*torch.Tensor*) – Input tensor

Returns

- **z_** (*torch.Tensor*) – Transformed tensor according to ActNorm layer with the shape of 'z'.
- **log_det** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

`inverse`(*z: Tensor*) → Tuple[Tensor, Tensor]

Inverse pass of the ActNorm layer

Parameters
z (*torch.Tensor*) – Input tensor

Returns

- **z_** (*torch.Tensor*) – Transformed tensor according to ActNorm layer with the shape of 'z'.
- **log_det** (*torch.Tensor*) – Tensor which represents the information of the deviation of the initial and target distribution.

class MLP_flow(*layers: list*, *leaky: float = 0.0*, *score_scale: float | None = None*, *output_fn=None*, *output_scale: float | None = None*, *init_zeros: bool = False*, *dropout: float | None = None*)

A Multi-Layer Perceptron (MLP) model for normalizing flows that is used as a part of a Normalizing Flow model. It is a modified version of the MLP model from *deepchem/deepchem/models/torch_models/layers.py* to handle multiple layers

Example

```
>>> import torch
>>> from deepchem.models.torch_models.flows import MLP_flow
>>> layers = [2, 4, 4, 2]
>>> mlp_flow = MLP_flow(layers)
>>> input = torch.tensor([1., 2.])
>>> output = mlp_flow(input)
>>> output.shape
torch.Size([2])
```

__init__(*layers: list*, *leaky: float = 0.0*, *score_scale: float | None = None*, *output_fn=None*, *output_scale: float | None = None*, *init_zeros: bool = False*, *dropout: float | None = None*)

Initializes the MLP_flow model

Parameters

- **layers** (*list*) – List of layer sizes from start to end
- **leaky** (*float, optional default 0.0*) – Slope of the leaky part of the ReLU, if 0.0, standard ReLU is used
- **score_scale** (*float, optional*) – Factor to apply to the scores, i.e. output before output_fn
- **output_fn** (*str, optional*) – Function to be applied to the output, either None, "sigmoid", "relu", "tanh", or "clampexp"
- **output_scale** (*float, optional*) – Rescale outputs if output_fn is specified, i.e. scale * output_fn(out / scale)
- **init_zeros** (*bool, optional*) – Flag, if true, weights and biases of last layer are initialized with zeros (helpful for deep models, see arXiv 1807.03039)
- **dropout** (*float, optional*) – If specified, dropout is done before last layer; if None, no dropout is done

forward(*x: Tensor*) → Tensor

Forward pass of the MLP_flow model

Parameters

x (*torch.Tensor*) – Input tensor

Returns

Transformed tensor according to the MLP_flow model with the shape of 'x'

Return type

torch.Tensor

Grover Layers

The following layers are used for implementing GROVER model as described in the paper <*Self-Supervised Graph Transformer on Large-Scale Molecular Data <https://drug.ai.tencent.com/publications/GROVER.pdf>*_

class **GroverMPNEncoder**(*atom_messages: bool*, *init_message_dim: int*, *hidden_size: int*, *depth: int*, *undirected: bool*, *attach_feats: bool*, *attached_feat_fdim: int = 0*, *bias: bool = True*, *dropout: float = 0.2*, *activation: str = 'relu'*, *input_layer: str = 'fc'*, *dynamic_depth: str = 'none'*)

Performs Message Passing to generate encodings for the molecule.

Parameters

- **atom_messages** (*bool*) – True if encoding atom-messages else False.
- **init_message_dim** (*int*) – Dimension of embedding message.
- **attach_feats** (*bool*) – Set to *True* if additional features are passed along with node/edge embeddings.
- **attached_feat_fdim** (*int*) – Dimension of additional features when *attach_feats* is *True*
- **undirected** (*bool*) – If set to *True*, the graph is considered as an undirected graph.
- **depth** (*int*) – number of hops in a message passing iteration
- **dynamic_depth** (*str, default: none*) – If set to *uniform* for randomly sampling dynamic depth from an uniform distribution else if set to *truncnorm*, dynamic depth is sampled from a truncated normal distribution.
- **input_layer** (*str*) – If set to *fc*, adds an initial feed-forward layer. If set to *none*, does not add an initial feed forward layer.

__init__(*atom_messages: bool*, *init_message_dim: int*, *hidden_size: int*, *depth: int*, *undirected: bool*, *attach_feats: bool*, *attached_feat_fdim: int = 0*, *bias: bool = True*, *dropout: float = 0.2*, *activation: str = 'relu'*, *input_layer: str = 'fc'*, *dynamic_depth: str = 'none'*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*init_messages*, *init_attached_features*, *a2nei*, *a2attached*, *b2a=None*, *b2revb=None*, *adjs=None*) → FloatTensor

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class **GroverAttentionHead**(*hidden_size: int = 128*, *bias: bool = True*, *depth: int = 1*, *dropout: float = 0.0*, *undirected: bool = False*, *atom_messages: bool = False*)

Generates attention head using GroverMPNEncoder for generating query, key and value

Parameters

- **hidden_size** (*int*) – Dimension of hidden layer
- **undirected** (*bool*) – If set to *True*, the graph is considered as an undirected graph.
- **depth** (*int*) – number of hops in a message passing iteration
- **atom_messages** (*bool*) – True if encoding atom-messages else False.

`__init__`(*hidden_size: int = 128*, *bias: bool = True*, *depth: int = 1*, *dropout: float = 0.0*, *undirected: bool = False*, *atom_messages: bool = False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*f_atoms*, *f_bonds*, *a2b*, *a2a*, *b2a*, *b2revb*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class GroverMTBlock`(*atom_messages: bool*, *input_dim: int*, *num_heads: int*, *depth: int*, *undirected: bool = False*, *hidden_size: int = 128*, *dropout: float = 0.0*, *bias: bool = True*, *res_connection: bool = True*, *activation: str = 'relu'*)

Message passing combined with transformer architecture

The layer combines message passing performed using GroverMPNEncoder and uses it to generate query, key and value for multi-headed Attention block.

Parameters

- **atom_messages** (*bool*) – True if encoding atom-messages else False.
- **input_dim** (*int*) – Dimension of input features
- **num_heads** (*int*) – Number of attention heads
- **depth** (*int*) – Number of hops in a message passing iteration
- **undirected** (*bool*) – If set to *True*, the graph is considered as an undirected graph.

`__init__`(*atom_messages: bool*, *input_dim: int*, *num_heads: int*, *depth: int*, *undirected: bool = False*, *hidden_size: int = 128*, *dropout: float = 0.0*, *bias: bool = True*, *res_connection: bool = True*, *activation: str = 'relu'*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*batch*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class GroverTransEncoder`(*node_fdim: int*, *edge_fdim: int*, *depth: int = 3*, *undirected: bool = False*, *num_mt_block: int = 2*, *num_heads: int = 2*, *hidden_size: int = 64*, *dropout: float = 0.2*, *res_connection: bool = True*, *bias: bool = True*, *activation: str = 'relu'*)

GroverTransEncoder for encoding a molecular graph

The GroverTransEncoder layer is used for encoding a molecular graph. The layer returns 4 outputs. They are atom messages aggregated from atom hidden states, atom messages aggregated from bond hidden states, bond messages aggregated from atom hidden states, bond messages aggregated from bond hidden states.

Parameters

- **hidden_size** (*int*) – the hidden size of the model.
- **edge_fdim** (*int*) – the dimension of additional feature for edge/bond.
- **node_fdim** (*int*) – the dimension of additional feature for node/atom.
- **depth** (*int*) – Dynamic message passing depth for use in MPNEncoder
- **undirected** (*bool*) – The message passing is undirected or not
- **dropout** (*float*) – the dropout ratio
- **activation** (*str*) – the activation function
- **num_mt_block** (*int*) – the number of mt block.
- **num_head** (*int*) – the number of attention AttentionHead.
- **bias** (*bool*) – enable bias term in all linear layers.
- **res_connection** (*bool*) – enables the skip-connection in MTBlock.

`__init__`(*node_fdim: int*, *edge_fdim: int*, *depth: int = 3*, *undirected: bool = False*, *num_mt_block: int = 2*, *num_heads: int = 2*, *hidden_size: int = 64*, *dropout: float = 0.2*, *res_connection: bool = True*, *bias: bool = True*, *activation: str = 'relu'*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*batch*)

Forward layer

Parameters
batch (*Tuple*) – A tuple of tensors representing grover attributes

Returns
embeddings – Embeddings for atom generated from hidden state of nodes and bonds and embeddings of bond generated from hidden states of nodes and bond.

Return type
Tuple[Tuple[torch.Tensor, torch.Tensor], Tuple[torch.Tensor, torch.Tensor]]

class GroverEmbedding(*node_fdim*, *edge_fdim*, *hidden_size=128*, *depth=1*, *undirected=False*, *dropout=0.2*, *activation='relu'*, *num_mt_block=1*, *num_heads=4*, *bias=False*, *res_connection=False*)

GroverEmbedding layer.

This layer is a simple wrapper over GroverTransEncoder layer for retrieving the embeddings from the GroverTransEncoder layer.

Parameters

- **edge_fdim** (*int*) – the dimension of additional feature for edge/bond.
- **node_fdim** (*int*) – the dimension of additional feature for node/atom.
- **depth** (*int*) – Dynamic message passing depth for use in MPNEncoder
- **undirected** (*bool*) – The message passing is undirected or not
- **num_mt_block** (*int*) – the number of message passing blocks.
- **num_head** (*int*) – the number of attention heads.

`__init__`(*node_fdim*, *edge_fdim*, *hidden_size=128*, *depth=1*, *undirected=False*, *dropout=0.2*, *activation='relu'*, *num_mt_block=1*, *num_heads=4*, *bias=False*, *res_connection=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*graph_batch: List[Tensor]*)

Forward function

Parameters

graph_batch (*List[torch.Tensor]*) – A list containing f_atoms, f_bonds, a2b, b2a, b2revb, a_scope, b_scope, a2a

Returns

embedding – Returns a dictionary of embeddings. The embeddings are: - atom_from_atom: node messages aggregated from node hidden states - bond_from_atom: bond messages aggregated from bond hidden states - atom_from_bond: node message aggregated from bond hidden states - bond_from_bond: bond messages aggregated from bond hidden states.

Return type

Dict[str, torch.Tensor]

class **GroverEmbedding**(*node_fdim*, *edge_fdim*, *hidden_size=128*, *depth=1*, *undirected=False*, *dropout=0.2*, *activation='relu'*, *num_mt_block=1*, *num_heads=4*, *bias=False*, *res_connection=False*)

GroverEmbedding layer.

This layer is a simple wrapper over GroverTransEncoder layer for retrieving the embeddings from the GroverTransEncoder layer.

Parameters

- **edge_fdim** (*int*) – the dimension of additional feature for edge/bond.
- **node_fdim** (*int*) – the dimension of additional feature for node/atom.
- **depth** (*int*) – Dynamic message passing depth for use in MPNEncoder
- **undirected** (*bool*) – The message passing is undirected or not
- **num_mt_block** (*int*) – the number of message passing blocks.
- **num_head** (*int*) – the number of attention heads.

__init__(*node_fdim*, *edge_fdim*, *hidden_size=128*, *depth=1*, *undirected=False*, *dropout=0.2*, *activation='relu'*, *num_mt_block=1*, *num_heads=4*, *bias=False*, *res_connection=False*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*graph_batch: List[Tensor]*)

Forward function

Parameters

graph_batch (*List[torch.Tensor]*) – A list containing f_atoms, f_bonds, a2b, b2a, b2revb, a_scope, b_scope, a2a

Returns

embedding – Returns a dictionary of embeddings. The embeddings are: - atom_from_atom: node messages aggregated from node hidden states - bond_from_atom: bond messages aggregated from bond hidden states - atom_from_bond: node message aggregated from bond hidden states - bond_from_bond: bond messages aggregated from bond hidden states.

Return type

Dict[str, torch.Tensor]

class **GroverAtomVocabPredictor**(*vocab_size: int*, *in_features: int = 128*)

Grover Atom Vocabulary Prediction Module.

The GroverAtomVocabPredictor module is used for predicting atom-vocabulary for the self-supervision task in Grover architecture. In the self-supervision tasks, one task is to learn contextual-information of nodes (atoms).

Contextual information are encoded as strings, like *C_N-DOUBLE1_O-SINGLE1*. The module accepts an atom encoding and learns to predict the contextual information of the atom as a multi-class classification problem.

Example

```
>>> from deepchem.models.torch_models.grover_layers import GroverAtomVocabPredictor
>>> num_atoms, in_features, vocab_size = 30, 16, 10
>>> layer = GroverAtomVocabPredictor(vocab_size, in_features)
>>> embedding = torch.randn(num_atoms, in_features)
>>> result = layer(embedding)
>>> result.shape
torch.Size([30, 10])
```

Reference

`__init__`(*vocab_size: int*, *in_features: int = 128*)

Initializing Grover Atom Vocabulary Predictor

Parameters

- **`vocab_size`** (*`int`*) – size of vocabulary (vocabulary here is the total number of different possible contexts)
- **`in_features`** (*`int`*) – feature size of atom embeddings.

`forward`(*embeddings*)

Parameters

`embeddings` (*`torch.Tensor`*) – the atom embeddings of shape (vocab_size, in_features)

Returns

logits – the prediction for each atom of shape (num_bond, vocab_size)

Return type

torch.Tensor

`class` **`GroverBondVocabPredictor`**(*vocab_size: int*, *in_features: int = 128*)

Layer for learning contextual information for bonds.

The layer is used in Grover architecture to learn contextual information of a bond by predicting the context of a bond from the bond embedding in a multi-class classification setting. The contextual information of a bond are encoded as strings (ex: '(DOUBLE-STEREONONE-NONE)_C-(SINGLE-STEREONONE-NONE)2').

Example

```
>>> from deepchem.models.torch_models.grover_layers import GroverBondVocabPredictor
>>> num_bonds = 20
>>> in_features, vocab_size = 16, 10
>>> layer = GroverBondVocabPredictor(vocab_size, in_features)
>>> embedding = torch.randn(num_bonds * 2, in_features)
>>> result = layer(embedding)
>>> result.shape
torch.Size([20, 10])
```

Reference

`__init__`(*vocab_size: int*, *in_features: int = 128*)

Initializes GroverBondVocabPredictor

Parameters

- **`vocab_size`** (*`int`*) – Size of vocabulary, used for number of classes in prediction.
- **`in_features`** (*`int, default: 128`*) – Input feature size of bond embeddings.

`forward`(*embeddings*)

Parameters

`embeddings` (*`torch.Tensor`*) – bond embeddings of shape (num_bond, in_features)

Returns

logits – the prediction for each bond, (num_bond, vocab_size)

Return type

torch.Tensor

`class` **`GroverFunctionalGroupPredictor`**(*functional_group_size: int*, *in_features=128*)

The functional group prediction task for self-supervised learning.

Molecules have functional groups in them. This module is used for predicting the functional group and the problem is formulated as an multi-label classification problem.

Parameters

- **`functional_group_size`** (*`int,`*) – size of functional group
- **`in_features`** (*`int,`*) – hidden_layer size, default 128

Example

```
>>> from deepchem.models.torch_models.grover_layers import␣
↪GroverFunctionalGroupPredictor
>>> in_features, functional_group_size = 8, 20
>>> num_atoms, num_bonds = 10, 20
>>> predictor = GroverFunctionalGroupPredictor(functional_group_size=20, in_
↪features=8)
>>> atom_scope, bond_scope = [(0, 3), (3, 3), (6, 4)], [(0, 5), (5, 4), (9, 11)]
>>> embeddings = {}
>>> embeddings['bond_from_atom'] = torch.randn(num_bonds, in_features)
>>> embeddings['bond_from_bond'] = torch.randn(num_bonds, in_features)
>>> embeddings['atom_from_atom'] = torch.randn(num_atoms, in_features)
>>> embeddings['atom_from_bond'] = torch.randn(num_atoms, in_features)
>>> result = predictor(embeddings, atom_scope, bond_scope)
```

Reference

`__init__`(*functional_group_size: int*, *in_features=128*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*embeddings: Dict*, *atom_scope: List*, *bond_scope: List*)

The forward function for the GroverFunctionalGroupPredictor (semantic motif prediction) layer. It takes atom/bond embeddings produced from node and bond hidden states from GroverEmbedding module and the atom, bond scopes and produces prediction logits for different each embedding. The scopes are used to differentiate atoms/bonds belonging to a molecule in a batched molecular graph.

Parameters

- **embedding** (*Dict*) – The input embeddings organized as an dictionary. The input embeddings are output of GroverEmbedding layer.
- **atom_scope** (*List*) – The scope for atoms.
- **bond_scope** (*List*) – The scope for bonds

Returns

- **preds** (*Dict*) – A dictionary containing the predicted logits of functional group from four different types of input embeddings. The key and their corresponding predictions
- *are described below.* –
  - atom_from_atom - prediction logits from atom embeddings generated via node hidden states
  - atom_from_bond - prediction logits from atom embeddings generated via bond hidden states
  - bond_from_atom - prediction logits from bond embeddings generated via node hidden states
  - bond_from_bond - prediction logits from bond embeddings generated via bond hidden states

class **GroverPretrain**(*embedding: Module*, *atom_vocab_task_atom: Module*, *atom_vocab_task_bond: Module*, *bond_vocab_task_atom: Module*, *bond_vocab_task_bond: Module*, *functional_group_predictor: Module*)

The Grover Pretrain module.

The GroverPretrain module is used for training an embedding based on the Grover Pretraining task. Grover pretraining is a self-supervised task where an embedding is trained to learn the contextual information of atoms and bonds along with graph-level properties, which are functional groups in case of molecular graphs.

Parameters

- **embedding** (*nn.Module*) – An embedding layer to generate embedding from input molecular graph
- **atom_vocab_task_atom** (*nn.Module*) – A layer used for predicting atom vocabulary from atom features generated via atom hidden states.
- **atom_vocab_task_bond** (*nn.Module*) – A layer used for predicting atom vocabulary from atom features generated via bond hidden states.
- **bond_vocab_task_atom** (*nn.Module*) – A layer used for predicting bond vocabulary from bond features generated via atom hidden states.

- **bond_vocab_task_bond** (*nn.Module*) – A layer used for predicting bond vocabulary from bond features generated via bond hidden states.

Returns

- **prediction_logits** (*Tuple*) – A tuple of prediction logits containing prediction logits of atom vocabulary task from atom hidden state,
- *prediction logits for atom vocabulary task from bond hidden states, prediction logits for bond vocabulary task*
- *from atom hidden states, prediction logits for bond vocabulary task from bond hidden states, functional*
- *group prediction logits from atom embedding generated from atom and bond hidden states, functional group*
- *prediction logits from bond embedding generated from atom and bond hidden states.*

Example

```
>>> import deepchem as dc
>>> from deepchem.utils.grover import BatchGroverGraph
>>> from deepchem.models.torch_models.grover import GroverPretrain
>>> from deepchem.models.torch_models.grover_layers import GroverEmbedding,␣
↪GroverAtomVocabPredictor, GroverBondVocabPredictor, GroverFunctionalGroupPredictor
>>> smiles = ['CC', 'CCC', 'CC(=O)C']
```

```
>>> fg = dc.feat.CircularFingerprint()
>>> featurizer = dc.feat.GroverFeaturizer(features_generator=fg)
```

```
>>> graphs = featurizer.featurize(smiles)
>>> batched_graph = BatchGroverGraph(graphs)
>>> grover_graph_attributes = batched_graph.get_components()
>>> f_atoms, f_bonds, a2b, b2a, b2revb, a2a, a_scope, b_scope, _ = grover_graph_
↪attributes
>>> components = {}
>>> components['embedding'] = GroverEmbedding(node_fdim=f_atoms.shape[1], edge_
↪fdim=f_bonds.shape[1])
>>> components['atom_vocab_task_atom'] = GroverAtomVocabPredictor(vocab_size=10, in_
↪features=128)
>>> components['atom_vocab_task_bond'] = GroverAtomVocabPredictor(vocab_size=10, in_
↪features=128)
>>> components['bond_vocab_task_atom'] = GroverBondVocabPredictor(vocab_size=10, in_
↪features=128)
>>> components['bond_vocab_task_bond'] = GroverBondVocabPredictor(vocab_size=10, in_
↪features=128)
>>> components['functional_group_predictor'] = GroverFunctionalGroupPredictor(10)
>>> model = GroverPretrain(**components)
```

```
>>> inputs = f_atoms, f_bonds, a2b, b2a, b2revb, a_scope, b_scope, a2a
>>> output = model(inputs)
```

Reference

`__init__`(*embedding: Module*, *atom_vocab_task_atom: Module*, *atom_vocab_task_bond: Module*, *bond_vocab_task_atom: Module*, *bond_vocab_task_bond: Module*, *functional_group_predictor: Module*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*graph_batch*)

Forward function

Parameters

`graph_batch` (*`List[torch.Tensor]`*) – A list containing grover graph attributes

`class GroverFinetune`(*embedding: Module*, *readout: Module*, *mol_atom_from_atom_ffn: Module*, *mol_atom_from_bond_ffn: Module*, *hidden_size: int = 128*, *mode: str = 'regression'*, *n_tasks: int = 1*, *n_classes: int | None = None*)

Grover Finetune model.

For a graph level prediction task, the GroverFinetune model uses node/edge embeddings output by the GroverEmbeddong layer and applies a readout function on it to get graph embeddings and use additional MLP layers to predict the property of the molecular graph.

Parameters

- **`embedding`** (*`nn.Module`*) – An embedding layer to generate embedding from input molecular graph
- **`readout`** (*`nn.Module`*) – A readout layer to perform readout atom and bond hidden states
- **`mol_atom_from_atom_ffn`** (*`nn.Module`*) – A feed forward network which learns representation from atom messages generated via atom hidden states of a molecular graph
- **`mol_atom_from_bond_ffn`** (*`nn.Module`*) – A feed forward network which learns representation from atom messages generated via bond hidden states of a molecular graph
- **`mode`** (*`str`*) – classification or regression

Returns

prediction_logits – prediction logits

Return type

torch.Tensor

Example

```
>>> import deepchem as dc
>>> from deepchem.utils.grover import BatchGroverGraph
>>> from deepchem.models.torch_models.grover_layers import GroverEmbedding
>>> from deepchem.models.torch_models.readout import GroverReadout
>>> from deepchem.models.torch_models.grover import GroverFinetune
>>> smiles = ['CC', 'CCC', 'CC(=O)C']
>>> fg = dc.feat.CircularFingerprint()
>>> featurizer = dc.feat.GroverFeaturizer(features_generator=fg)
>>> graphs = featurizer.featurize(smiles)
>>> batched_graph = BatchGroverGraph(graphs)
>>> attributes = batched_graph.get_components()
>>> components = {}
```

(continues on next page)

(continued from previous page)

```
>>> additional_features = batched_graph.additional_features
>>> f_atoms, f_bonds, a2b, b2a, b2revb, a2a, a_scope, b_scope, fg_labels =␣
↪attributes
>>> inputs = f_atoms, f_bonds, a2b, b2a, b2revb, a_scope, b_scope, a2a
>>> components = {}
>>> components['embedding'] = GroverEmbedding(node_fdim=f_atoms.shape[1], edge_
↪fdim=f_bonds.shape[1])
>>> components['readout'] = GroverReadout(rtype="mean", in_features=128)
>>> components['mol_atom_from_atom_ffn'] = nn.Linear(in_features=additional_
↪features.shape[1]+ 128, out_features=128)
>>> components['mol_atom_from_bond_ffn'] = nn.Linear(in_features=additional_
↪features.shape[1] + 128, out_features=128)
>>> model = GroverFinetune(**components, mode='regression', hidden_size=128)
>>> model.training = False
>>> output = model((inputs, additional_features))
```

Reference

`__init__`(*embedding: Module*, *readout: Module*, *mol_atom_from_atom_ffn: Module*, *mol_atom_from_bond_ffn: Module*, *hidden_size: int = 128*, *mode: str = 'regression'*, *n_tasks: int = 1*, *n_classes: int | None = None*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*inputs*)

Parameters
`inputs` (*`Tuple`*) – grover batch graph attributes

Attention Layers

class `ScaledDotProductAttention`

The Scaled Dot Production Attention operation from *Attention Is All You Need <https://arxiv.org/abs/1706.03762>*_ paper.

Example

```
>>> from deepchem.models import ScaledDotProductAttention as SDPA
>>> attn = SDPA()
>>> x = torch.ones(1, 5)
>>> # Linear layers for making query, key, value
>>> Q, K, V = nn.Parameter(torch.ones(5)), nn.Parameter(torch.ones(5)), nn.
↪Parameter(torch.ones(5))
>>> query, key, value = Q * x, K * x, V * x
>>> x_out, attn_score = attn(query, key, value)
```

`__init__`()

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*query: Tensor*, *key: Tensor*, *value: Tensor*, *mask: Tensor | None = None*, *dropout: Dropout | None = None*)

Parameters

- **query** (*torch.Tensor*) – Query tensor for attention
- **key** (*torch.Tensor*) – Key tensor for attention
- **value** (*torch.Tensor*) – Value tensor for attention
- **mask** (*torch.Tensor (optional)*) – Mask to apply during attention computation
- **dropout** (*nn.Dropout (optional)*) – Dropout layer for attention output

class **SelfAttention**(*in_features*, *out_features*, *hidden_size=128*)

SelfAttention Layer

Given $Xin mathbb{R}^{n imes in_feature}$, the attention is calculated by: $a=softmax(W_2tanh(W_1X))$, where $W_1 in mathbb{R}^{hidden imes in_feature}$, $W_2 in mathbb{R}^{out_feature imes hidden}$. The final output is $y=aX$ where $y in mathbb{R}^{n imes out_feature}$.

Parameters

- **in_features** (*int*) – Dimension of input features
- **out_features** (*int*) – Dimension of output features
- **hidden_size** (*int*) – Dimension of hidden layer

__init__(*in_features*, *out_features*, *hidden_size=128*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*X*)

The forward function.

Parameters

X (*torch.Tensor*) – input feature of shape $mathbb{R}^{n imes in_feature}$.

Returns

- **embedding** (*torch.Tensor*) – The final embedding of shape $mathbb{R}^{out_features imes in_feature}$
- **attention-matrix** (*torch.Tensor*) – The attention matrix

Readout Layers

class **GroverReadout**(*rtype: str = 'mean'*, *in_features: int = 128*, *attn_hidden_size: int = 32*, *attn_out_size: int = 32*)

Performs readout on a batch of graph

The readout module is used for performing readouts on batched graphs to convert node embeddings/edge embeddings into graph embeddings. It is used in the Grover architecture to generate a graph embedding from node and edge embeddings. The generate embedding can be used in downstream tasks like graph classification or graph prediction problems.

Parameters

- **rtype** (*str*) – Readout type, can be 'mean' or 'self-attention'
- **in_features** (*int*) – Size fof input features

- **attn_hidden_size** (*int*) – If readout type is attention, size of hidden layer in attention network.
- **attn_out_size** (*int*) – If readout type is attention, size of attention out layer.

Example

```
>>> import torch
>>> from deepchem.models.torch_models.readout import GroverReadout
>>> n_nodes, n_features = 6, 32
>>> readout = GroverReadout(rtype="mean")
>>> embedding = torch.ones(n_nodes, n_features)
>>> result = readout(embedding, scope=[(0, 6)])
>>> result.size()
torch.Size([1, 32])
```

__init__(*rtype: str = 'mean'*, *in_features: int = 128*, *attn_hidden_size: int = 32*, *attn_out_size: int = 32*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*graph_embeddings: Tensor*, *scope: List[List]*) → Tensor

Given a batch node/edge embedding and a scope list, produce the graph-level embedding by scope.

Parameters

- **embeddings** (*torch.Tensor*) – The embedding matrix, num_nodes x in_features or num_edges x in_features.
- **scope** (*List[List]*) – A list, in which the element is a list [start, range]. *start* is the index, *range* is the length of scope. (start + range = end)

Returns

graph_embeddings – A stacked tensor containing graph embeddings of shape len(scope) x in_features if readout type is mean or len(scope) x attn_out_size when readout type is self-attention.

Return type

torch.Tensor

3.23.4 Jax Layers

class Linear(*num_output: int*, *initializer: str = 'linear'*, *use_bias: bool = True*, *bias_init: float = 0.0*, *name: str = 'linear'*)

Protein folding specific Linear Module.

This differs from the standard Haiku Linear in a few ways:

- It supports inputs of arbitrary rank
- Initializers are specified by strings

This code is adapted from DeepMind's AlphaFold code release (https://github.com/deepmind/alphafold).

Examples

```
>>> import deepchem as dc
>>> import haiku as hk
>>> import jax
>>> import deepchem.models.jax_models.layers
>>> def forward_model(x):
...   layer = dc.models.jax_models.layers.Linear(2)
...   return layer(x)
>>> f = hk.transform(forward_model)
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.ones([8, 28 * 28])
>>> params = f.init(rng, x)
>>> output = f.apply(params, rng, x)
```

`__init__`(*num_output: int*, *initializer: str = 'linear'*, *use_bias: bool = True*, *bias_init: float = 0.0*, *name: str = 'linear'*)

Constructs Linear Module.

Parameters

- **`num_output`** (*`int`*) – number of output channels.
- **`initializer`** (*`str (default 'linear')`*) – What initializer to use, should be one of {'linear', 'relu', 'zeros'}
- **`use_bias`** (*`bool (default True)`*) – Whether to include trainable bias
- **`bias_init`** (*`float (default 0)`*) – Value used to initialize bias.
- **`name`** (*`str (default 'linear')`*) – name of module, used for name scopes.

3.23.5 Density Functional Theory Layers

3.24 Metrics

Metrics are one of the most important parts of machine learning. Unlike traditional software, in which algorithms either work or don't work, machine learning models work in degrees. That is, there's a continuous range of "goodness" for a model. "Metrics" are functions which measure how well a model works. There are many different choices of metrics depending on the type of model at hand.

3.24.1 Metric Utilities

Metric utility functions allow for some common manipulations such as switching to/from one-hot representations.

`to_one_hot`(*y: ndarray*, *n_classes: int = 2*) → ndarray

Transforms label vector into one-hot encoding.

Turns y into vector of shape *(N, n_classes)* with a one-hot encoding. Assumes that *y* takes values from *0* to *n_classes - 1*.

Parameters

- **y** (*`np.ndarray`*) – A vector of shape *(N,)* or *(N, 1)*

- **n_classes** (*int, default 2*) – If specified use this as the number of classes. Else will try to impute it as *n_classes = max(y) + 1* for arrays and as *n_classes=2* for the case of scalars. Note this parameter only has value if *mode=="classification"*

Returns
A numpy array of shape *(N, n_classes)*.

Return type
np.ndarray

from_one_hot(*y: ndarray*, *axis: int = 1*) → ndarray

Transforms label vector from one-hot encoding.

Parameters

- **y** (*np.ndarray*) – A vector of shape *(n_samples, num_classes)*
- **axis** (*int, optional (default 1)*) – The axis with one-hot encodings to reduce on.

Returns
A numpy array of shape *(n_samples,)*

Return type
np.ndarray

3.24.2 Metric Shape Handling

One of the trickiest parts of handling metrics correctly is making sure the shapes of input weights, predictions and labels and processed correctly. This is challenging in particular since DeepChem supports multitask, multiclass models which means that shapes must be handled with care to prevent errors. DeepChem maintains the following utility functions which attempt to facilitate shape handling for you.

normalize_weight_shape(*w: ndarray | None*, *n_samples: int*, *n_tasks: int*) → ndarray

A utility function to correct the shape of the weight array.

This utility function is used to normalize the shapes of a given weight array.

Parameters

- **w** (*np.ndarray*) – *w* can be *None* or a scalar or a *np.ndarray* of shape *(n_samples,)* or of shape *(n_samples, n_tasks)*. If *w* is a scalar, it's assumed to be the same weight for all samples/tasks.
- **n_samples** (*int*) – The number of samples in the dataset. If *w* is not None, we should have *n_samples = w.shape[0]* if *w* is a ndarray
- **n_tasks** (*int*) – The number of tasks. If *w* is 2d ndarray, then we should have *w.shape[1] == n_tasks*.

Examples

```
>>> import numpy as np
>>> w_out = normalize_weight_shape(None, n_samples=10, n_tasks=1)
>>> (w_out == np.ones((10, 1))).all()
True
```

Returns
w_out – Array of shape *(n_samples, n_tasks)*

Return type
np.ndarray

normalize_labels_shape(*y: ndarray*, *mode: str | None = None*, *n_tasks: int | None = None*, *n_classes: int | None = None*) → ndarray

A utility function to correct the shape of the labels.

Parameters

- **y** (*np.ndarray*) – *y* is an array of shape *(N,)* or *(N, n_tasks)* or *(N, n_tasks, 1)*.
- **mode** (*str, default None*) – If *mode* is "classification" or "regression", attempts to apply data transformations.
- **n_tasks** (*int, default None*) – The number of tasks this class is expected to handle.
- **n_classes** (*int, default None*) – If specified use this as the number of classes. Else will try to impute it as *n_classes = max(y) + 1* for arrays and as *n_classes=2* for the case of scalars. Note this parameter only has value if *mode=="classification"*

Returns
y_out – If *mode=="classification"*, *y_out* is an array of shape *(N, n_tasks, n_classes)*. If *mode=="regression"*, *y_out* is an array of shape *(N, n_tasks)*.

Return type
np.ndarray

normalize_prediction_shape(*y: ndarray*, *mode: str | None = None*, *n_tasks: int | None = None*, *n_classes: int | None = None*)

A utility function to correct the shape of provided predictions.

The metric computation classes expect that inputs for classification have the uniform shape *(N, n_tasks, n_classes)* and inputs for regression have the uniform shape *(N, n_tasks)*. This function normalizes the provided input array to have the desired shape.

Examples

```
>>> import numpy as np
>>> y = np.random.rand(10)
>>> y_out = normalize_prediction_shape(y, "regression", n_tasks=1)
>>> y_out.shape
(10, 1)
```

Parameters

- **y** (*np.ndarray*) – If *mode=="classification"*, *y* is an array of shape *(N,)* or *(N, n_tasks)* or *(N, n_tasks, n_classes)*. If *mode=="regression"*, *y* is an array of shape *(N,)* or *(N, n_tasks)`or `(N, n_tasks, 1)*.
- **mode** (*str, default None*) – If *mode* is "classification" or "regression", attempts to apply data transformations.
- **n_tasks** (*int, default None*) – The number of tasks this class is expected to handle.
- **n_classes** (*int, default None*) – If specified use this as the number of classes. Else will try to impute it as *n_classes = max(y) + 1* for arrays and as *n_classes=2* for the case of scalars. Note this parameter only has value if *mode=="classification"*

Returns
  y_out – If *mode==”classification”*, *y_out* is an array of shape *(N, n_tasks, n_classes)*. If *mode==”regression”*, *y_out* is an array of shape *(N, n_tasks)*.

Return type
  np.ndarray

handle_classification_mode(*y: ndarray*, *classification_handling_mode: str | None*, *threshold_value: float | None = None*) → ndarray

Handle classification mode.

Transform predictions so that they have the correct classification mode.

Parameters

- **y** (*np.ndarray*) – Must be of shape *(N, n_tasks, n_classes)*
- **classification_handling_mode** (*str, default None*) – DeepChem models by default predict class probabilities for classification problems. This means that for a given singletask prediction, after shape normalization, the DeepChem prediction will be a numpy array of shape *(N, n_classes)* with class probabilities. *classification_handling_mode* is a string that instructs this method how to handle transforming these probabilities. It can take on the following values: - None: default value. Pass in *y_pred* directy into *self.metric*. - “threshold”: Use *threshold_predictions* to threshold *y_pred*. Use

  threshold_value as the desired threshold.

  – ”threshold-one-hot”: Use *threshold_predictions* to threshold *y_pred* using *threshold_values*, then apply *to_one_hot* to output.

- **threshold_value** (*float, default None*) – If set, and *classification_handling_mode* is “threshold” or “threshold-one-hot” apply a thresholding operation to values with this threshold. This option isj only sensible on binary classification tasks. If float, this will be applied as a binary classification value.

Returns
  y_out – If *classification_handling_mode* is “direct”, then of shape *(N, n_tasks, n_classes)*. If *classification_handling_mode* is “threshold”, then of shape *(N, n_tasks)*. If `classification_handling_mode is “threshold-one-hot”, then of shape `(N, n_tasks, n_classes)”

Return type
  np.ndarray

3.24.3 Metric Functions

DeepChem has a variety of different metrics which are useful for measuring model performance. A number (but not all) of these metrics are directly sourced from `sklearn`.

matthews_corrcoef(*y_true*, *y_pred*, ***, *sample_weight=None*)

Compute the Matthews correlation coefficient (MCC).

The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary and multiclass classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient. [source: Wikipedia]

Binary and multiclass labels are supported. Only in the binary case does this relate to information about true and false positives and negatives. See references below.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,)*) – Ground truth (correct) target values.
- **y_pred** (*array-like of shape (n_samples,)*) – Estimated targets as returned by a classifier.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

  New in version 0.18.

Returns

mcc – The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 and inverse prediction).

Return type

float

References

Examples

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

recall_score(*y_true*, *y_pred*, ***, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the recall.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

Support beyond term:*binary* targets is achieved by treating multiclass and multilabel data as a collection of binary problems, one for each label. For the binary case, setting *average='binary'* will return recall for *pos_label*. If *average* is not *'binary'*, *pos_label* is ignored and recall for both classes are computed then averaged or both returned (when *average=None*). Similarly, for multiclass and multilabel targets, recall for all *labels* are either returned or averaged depending on the *average* parameter. Use *labels* specify the set of labels to calculate recall for.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier.

- **labels** (*array-like, default=None*) – The set of labels to include when *average != 'binary'*, and their order if *average is None*. Labels present in the data can be excluded, for example in multiclass classification to exclude a "negative class". Labels not present in the data can be included and will be "assigned" 0 samples. For multilabel targets, labels are column indices. By default, all labels in *y_true* and *y_pred* are used in sorted order.

  Changed in version 0.17: Parameter *labels* improved for multiclass problem.

- **pos_label** (*int, float, bool or str, default=1*) – The class to report if *average='binary'* and the data is binary, otherwise this parameter is ignored. For multiclass or multilabel targets, set *labels=[pos_label]* and *average != 'binary'* to report metrics for one label only.

- **average** (*{'micro', 'macro', 'samples', 'weighted', 'binary'} or None, default='binary'*) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

  'binary':
  Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

  'micro':
  Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':
  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted':
  Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall. Weighted recall is equal to accuracy.

  'samples':
  Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

- **zero_division** (*{"warn", 0.0, 1.0, np.nan}, default="warn"*) – Sets the value to return when there is a zero division.

  Notes: - If set to "warn", this acts like 0, but a warning is also raised. - If set to *np.nan*, such values will be excluded from the average.

  New in version 1.3: *np.nan* option was added.

Returns
recall – Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

Return type
float (if average is not None) or array of float of shape (n_unique_labels,)

See also:

precision_recall_fscore_support
Compute precision, recall, F-measure and support for each class.

precision_score
Compute the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives.

balanced_accuracy_score
Compute balanced accuracy to deal with imbalanced datasets.

multilabel_confusion_matrix
Compute a confusion matrix for each class or sample.

PrecisionRecallDisplay.from_estimator
Plot precision-recall curve given an estimator and some data.

PrecisionRecallDisplay.from_predictions
Plot precision-recall curve given binary class predictions.

Notes

When `true positive + false negative == 0`, recall returns 0 and raises `UndefinedMetricWarning`. This behavior can be modified with `zero_division`.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import recall_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> recall_score(y_true, y_pred, average='macro')
0.33...
>>> recall_score(y_true, y_pred, average='micro')
0.33...
>>> recall_score(y_true, y_pred, average='weighted')
0.33...
>>> recall_score(y_true, y_pred, average=None)
array([1., 0., 0.])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> recall_score(y_true, y_pred, average=None)
array([0.5, 0. , 0. ])
>>> recall_score(y_true, y_pred, average=None, zero_division=1)
array([0.5, 1. , 1. ])
>>> recall_score(y_true, y_pred, average=None, zero_division=np.nan)
array([0.5, nan, nan])
```

```
>>> # multilabel classification
>>> y_true = [[0, 0, 0], [1, 1, 1], [0, 1, 1]]
>>> y_pred = [[0, 0, 0], [1, 1, 1], [1, 1, 0]]
>>> recall_score(y_true, y_pred, average=None)
array([1. , 1. , 0.5])
```

r2_score(*y_true*, *y_pred*, ***, *sample_weight=None*, *multioutput='uniform_average'*, *force_finite=True*)
R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). In the general case when the true y is non-constant, a constant model that always predicts the average y disregarding the input features would get a R^2 score of 0.0.

In the particular case when `y_true` is constant, the R^2 score is not finite: it is either `NaN` (perfect predictions) or `-Inf` (imperfect predictions). To prevent such non-finite numbers to pollute higher-level experiments such as a grid search cross-validation, by default these cases are replaced with 1.0 (perfect predictions) or 0.0 (imperfect predictions) respectively. You can set `force_finite` to `False` to prevent this fix from happening.

Note: when the prediction residuals have zero mean, the R^2 score is identical to the `Explained Variance score`.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Ground truth (correct) target values.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Estimated target values.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **multioutput** (*{'raw_values', 'uniform_average', 'variance_weighted'}, array-like of shape (n_outputs,) or None, default='uniform_average'*) – Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is "uniform_average".

  'raw_values' :
  Returns a full set of scores in case of multioutput input.

  'uniform_average' :
  Scores of all outputs are averaged with uniform weight.

  'variance_weighted' :
  Scores of all outputs are averaged, weighted by the variances of each individual output.

  Changed in version 0.19: Default value of multioutput is 'uniform_average'.

- **force_finite** (*bool, default=True*) – Flag indicating if `NaN` and `-Inf` scores resulting from constant data should be replaced with real numbers (`1.0` if prediction is perfect, `0.0` otherwise). Default is `True`, a convenient setting for hyperparameters' search procedures (e.g. grid search cross-validation).

  New in version 1.1.

Returns

z – The R^2 score or ndarray of scores if 'multioutput' is 'raw_values'.

Return type

float or ndarray of floats

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

This metric is not well-defined for single samples and will return a NaN value if n_samples is less than two.

References

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred,
...          multioutput='variance_weighted')
0.938...
>>> y_true = [1, 2, 3]
>>> y_pred = [1, 2, 3]
>>> r2_score(y_true, y_pred)
1.0
>>> y_true = [1, 2, 3]
>>> y_pred = [2, 2, 2]
>>> r2_score(y_true, y_pred)
0.0
>>> y_true = [1, 2, 3]
>>> y_pred = [3, 2, 1]
>>> r2_score(y_true, y_pred)
-3.0
>>> y_true = [-2, -2, -2]
>>> y_pred = [-2, -2, -2]
>>> r2_score(y_true, y_pred)
1.0
>>> r2_score(y_true, y_pred, force_finite=False)
nan
>>> y_true = [-2, -2, -2]
>>> y_pred = [-2, -2, -2 + 1e-8]
>>> r2_score(y_true, y_pred)
0.0
>>> r2_score(y_true, y_pred, force_finite=False)
-inf
```

mean_squared_error(*y_true*, *y_pred*, ***, *sample_weight=None*, *multioutput='uniform_average'*, *squared='deprecated'*)

Mean squared error regression loss.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Ground truth (correct) target values.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Estimated target values.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **multioutput** (*{'raw_values', 'uniform_average'} or array-like of shape (n_outputs,), default='uniform_average'*) – Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

  'raw_values' :
  Returns a full set of errors in case of multioutput input.

  'uniform_average' :
  Errors of all outputs are averaged with uniform weight.
- **squared** (*bool, default=True*) – If True returns MSE value, if False returns RMSE value.

  Deprecated since version 1.4: *squared* is deprecated in 1.4 and will be removed in 1.6. Use `root_mean_squared_error()` instead to calculate the root mean squared error.

Returns
loss – A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Return type
float or ndarray of floats

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
array([0.41666667, 1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.825...
```

mean_absolute_error(*y_true*, *y_pred*, ***, *sample_weight=None*, *multioutput='uniform_average'*)

Mean absolute error regression loss.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Ground truth (correct) target values.

- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Estimated target values.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **multioutput** (*{'raw_values', 'uniform_average'} or array-like of shape (n_outputs,), default='uniform_average'*) – Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

  'raw_values' :
  Returns a full set of errors in case of multioutput input.

  'uniform_average' :
  Errors of all outputs are averaged with uniform weight.

Returns

loss – If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

MAE output is non-negative floating point. The best value is 0.0.

Return type

float or ndarray of floats

Examples

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85...
```

precision_score(*y_true*, *y_pred*, ***, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the precision.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Support beyond term:*binary* targets is achieved by treating multiclass and multilabel data as a collection of binary problems, one for each label. For the binary case, setting *average='binary'* will return precision for *pos_label*. If *average* is not *'binary'*, *pos_label* is ignored and precision for both classes are computed, then averaged or both returned (when *average=None*). Similarly, for multiclass and multilabel targets, precision for all *labels* are either returned or averaged depending on the *average* parameter. Use *labels* specify the set of labels to calculate precision for.

Read more in the User Guide.

Parameters

- `y_true` (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- `y_pred` (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier.
- `labels` (*array-like, default=None*) – The set of labels to include when *average != 'binary'*, and their order if *average is None*. Labels present in the data can be excluded, for example in multiclass classification to exclude a "negative class". Labels not present in the data can be included and will be "assigned" 0 samples. For multilabel targets, labels are column indices. By default, all labels in *y_true* and *y_pred* are used in sorted order.

  Changed in version 0.17: Parameter *labels* improved for multiclass problem.
- `pos_label` (*int, float, bool or str, default=1*) – The class to report if *average='binary'* and the data is binary, otherwise this parameter is ignored. For multiclass or multilabel targets, set *labels=[pos_label]* and *average != 'binary'* to report metrics for one label only.
- `average` (*{'micro', 'macro', 'samples', 'weighted', 'binary'} or None, default='binary'*) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

  `'binary'`:
  Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

  `'micro'`:
  Calculate metrics globally by counting the total true positives, false negatives and false positives.

  `'macro'`:
  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  `'weighted'`:
  Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

  `'samples'`:
  Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).
- `sample_weight` (*array-like of shape (n_samples,), default=None*) – Sample weights.
- `zero_division` (*{"warn", 0.0, 1.0, np.nan}, default="warn"*) – Sets the value to return when there is a zero division.

  Notes: - If set to "warn", this acts like 0, but a warning is also raised. - If set to *np.nan*, such values will be excluded from the average.

  New in version 1.3: *np.nan* option was added.

Returns
: **precision** – Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

Return type
: float (if average is not None) or array of float of shape (n_unique_labels,)

See also:

precision_recall_fscore_support
: Compute precision, recall, F-measure and support for each class.

recall_score
: Compute the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives.

PrecisionRecallDisplay.from_estimator
: Plot precision-recall curve given an estimator and some data.

PrecisionRecallDisplay.from_predictions
: Plot precision-recall curve given binary class predictions.

multilabel_confusion_matrix
: Compute a confusion matrix for each class or sample.

Notes

When `true positive + false positive == 0`, precision returns 0 and raises `UndefinedMetricWarning`. This behavior can be modified with `zero_division`.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> precision_score(y_true, y_pred, average='macro')
0.22...
>>> precision_score(y_true, y_pred, average='micro')
0.33...
>>> precision_score(y_true, y_pred, average='weighted')
0.22...
>>> precision_score(y_true, y_pred, average=None)
array([0.66..., 0.        , 0.        ])
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> precision_score(y_true, y_pred, average=None)
array([0.33..., 0.        , 0.        ])
>>> precision_score(y_true, y_pred, average=None, zero_division=1)
array([0.33..., 1.        , 1.        ])
>>> precision_score(y_true, y_pred, average=None, zero_division=np.nan)
array([0.33...,        nan,        nan])
```

```
>>> # multilabel classification
>>> y_true = [[0, 0, 0], [1, 1, 1], [0, 1, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> y_pred = [[0, 0, 0], [1, 1, 1], [1, 1, 0]]
>>> precision_score(y_true, y_pred, average=None)
array([0.5, 1. , 1. ])
```

`precision_recall_curve`(*y_true*, *probas_pred*, *, *pos_label=None*, *sample_weight=None*, *drop_intermediate=False*)

Compute precision-recall pairs for different probability thresholds.

Note: this implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the y axis.

The first precision and recall values are precision=class balance and recall=1.0 which corresponds to a classifier that always predicts the positive class.

Read more in the User Guide.

Parameters

- **`y_true`** (*`array-like of shape (n_samples,)`*) – True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.
- **`probas_pred`** (*`array-like of shape (n_samples,)`*) – Target scores, can either be probability estimates of the positive class, or non-thresholded measure of decisions (as returned by *decision_function* on some classifiers).
- **`pos_label`** (*`int, float, bool or str, default=None`*) – The label of the positive class. When `pos_label=None`, if y_true is in {-1, 1} or {0, 1}, `pos_label` is set to 1, otherwise an error will be raised.
- **`sample_weight`** (*`array-like of shape (n_samples,), default=None`*) – Sample weights.
- **`drop_intermediate`** (*`bool, default=False`*) – Whether to drop some suboptimal thresholds which would not appear on a plotted precision-recall curve. This is useful in order to create lighter precision-recall curves.

  New in version 1.3.

Returns

- **precision** (*ndarray of shape (n_thresholds + 1,)*) – Precision values such that element i is the precision of predictions with score >= thresholds[i] and the last element is 1.
- **recall** (*ndarray of shape (n_thresholds + 1,)*) – Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0.
- **thresholds** (*ndarray of shape (n_thresholds,)*) – Increasing thresholds on the decision function used to compute precision and recall where *n_thresholds = len(np.unique(probas_pred))*.

See also:

PrecisionRecallDisplay.from_estimator
Plot Precision Recall Curve given a binary classifier.

PrecisionRecallDisplay.from_predictions
Plot Precision Recall Curve using predictions from a binary classifier.

average_precision_score
Compute average precision from prediction scores.

det_curve
Compute error rates for different probability thresholds.

roc_curve
Compute Receiver operating characteristic (ROC) curve.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> precision, recall, thresholds = precision_recall_curve(
...     y_true, y_scores)
>>> precision
array([0.5       , 0.66666667, 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 1. , 0.5, 0.5, 0. ])
>>> thresholds
array([0.1 , 0.35, 0.4 , 0.8 ])
```

auc(*x*, *y*)

Compute Area Under the Curve (AUC) using the trapezoidal rule.

This is a general function, given points on a curve. For computing the area under the ROC-curve, see *roc_auc_score()*. For an alternative way to summarize a precision-recall curve, see average_precision_score().

Parameters

- **x** (*array-like of shape (n,)*) – X coordinates. These must be either monotonic increasing or monotonic decreasing.
- **y** (*array-like of shape (n,)*) – Y coordinates.

Returns
auc – Area Under the Curve.

Return type
float

See also:

roc_auc_score
Compute the area under the ROC curve.

average_precision_score
Compute average precision from prediction scores.

precision_recall_curve
Compute precision-recall pairs for different probability thresholds.

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

jaccard_score(*y_true*, *y_pred*, ***, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Jaccard similarity coefficient score.

The Jaccard index [1], or Jaccard similarity coefficient, defined as the size of the intersection divided by the size of the union of two label sets, is used to compare set of predicted labels for a sample to the corresponding set of labels in `y_true`.

Support beyond term:*binary* targets is achieved by treating multiclass and multilabel data as a collection of binary problems, one for each label. For the binary case, setting *average='binary'* will return the Jaccard similarity coefficient for *pos_label*. If *average* is not *'binary'*, *pos_label* is ignored and scores for both classes are computed, then averaged or both returned (when *average=None*). Similarly, for multiclass and multilabel targets, scores for all *labels* are either returned or averaged depending on the *average* parameter. Use *labels* specify the set of labels to calculate the score for.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) labels.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted labels, as returned by a classifier.
- **labels** (*array-like of shape (n_classes,), default=None*) – The set of labels to include when *average != 'binary'*, and their order if *average is None*. Labels present in the data can be excluded, for example in multiclass classification to exclude a "negative class". Labels not present in the data can be included and will be "assigned" 0 samples. For multilabel targets, labels are column indices. By default, all labels in *y_true* and *y_pred* are used in sorted order.
- **pos_label** (*int, float, bool or str, default=1*) – The class to report if *average='binary'* and the data is binary, otherwise this parameter is ignored. For multiclass or multilabel targets, set *labels=[pos_label]* and *average != 'binary'* to report metrics for one label only.
- **average** (*{'micro', 'macro', 'samples', 'weighted', 'binary'} or None, default='binary'*) – If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

  'binary':
  Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro':
Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro':
Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted':
Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance.

'samples':
Calculate metrics for each instance, and find their average (only meaningful for multilabel classification).

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **zero_division** (*"warn", {0.0, 1.0}, default="warn"*) – Sets the value to return when there is a zero division, i.e. when there there are no negative values in predictions and labels. If set to "warn", this acts like 0, but a warning is also raised.

Returns
score – The Jaccard score. When *average* is not *None*, a single scalar is returned.

Return type
float or ndarray of shape (n_unique_labels,), dtype=np.float64

See also:

accuracy_score
Function for calculating the accuracy score.

f1_score
Function for calculating the F1 score.

multilabel_confusion_matrix
Function for computing a confusion matrix for each class or sample.

Notes

jaccard_score() may be a poor metric if there are no positives for some samples or classes. Jaccard is undefined if there are no true or predicted labels, and our implementation will return a score of 0 with a warning.

References

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                    [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                    [1, 0, 0]])
```

In the binary case:

```
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the 2D comparison case (e.g. image similarity):

```
>>> jaccard_score(y_true, y_pred, average="micro")
0.6
```

In the multilabel case:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1. ])
```

In the multiclass case:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
array([1. , 0. , 0.33...])
```

f1_score(*y_true*, *y_pred*, ***, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*, *zero_division='warn'*)

Compute the F1 score, also known as balanced F-score or F-measure.

The F1 score can be interpreted as a harmonic mean of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

$$\text{F1} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

Where TP is the number of true positives, FN is the number of false negatives, and FP is the number of false positives. F1 is by default calculated as 0.0 when there are no true positives, false negatives, or false positives.

Support beyond binary targets is achieved by treating multiclass and multilabel data as a collection of binary problems, one for each label. For the binary case, setting *average='binary'* will return F1 score for *pos_label*. If *average* is not *'binary'*, *pos_label* is ignored and F1 score for both classes are computed, then averaged or both returned (when *average=None*). Similarly, for multiclass and multilabel targets, F1 score for all *labels* are either returned or averaged depending on the *average* parameter. Use *labels* specify the set of labels to calculate F1 score for.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier.
- **labels** (*array-like, default=None*) – The set of labels to include when *average != 'binary'*, and their order if *average is None*. Labels present in the data can be excluded, for example in multiclass classification to exclude a "negative class". Labels not present in the

data can be included and will be "assigned" 0 samples. For multilabel targets, labels are column indices. By default, all labels in *y_true* and *y_pred* are used in sorted order.

Changed in version 0.17: Parameter *labels* improved for multiclass problem.

- **pos_label** (*int, float, bool or str, default=1*) – The class to report if *average='binary'* and the data is binary, otherwise this parameter is ignored. For multiclass or multilabel targets, set *labels=[pos_label]* and *average != 'binary'* to report metrics for one label only.
- **average** (*{'micro', 'macro', 'samples', 'weighted', 'binary'} or None, default='binary'*) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

  'binary':
  Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

  'micro':
  Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':
  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted':
  Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

  'samples':
  Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **zero_division** (*{"warn", 0.0, 1.0, np.nan}, default="warn"*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative.

  Notes: - If set to "warn", this acts like 0, but a warning is also raised. - If set to *np.nan*, such values will be excluded from the average.

  New in version 1.3: *np.nan* option was added.

Returns
f1_score – F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

Return type
float or array of float, shape = [n_unique_labels]

See also:

`fbeta_score`
Compute the F-beta score.

`precision_recall_fscore_support`
Compute the precision, recall, F-score, and support.

jaccard_score
Compute the Jaccard similarity coefficient score.

multilabel_confusion_matrix
Compute a confusion matrix for each class or sample.

Notes

When `true positive + false positive + false negative == 0` (i.e. a class is completely absent from both `y_true` or `y_pred`), f-score is undefined. In such cases, by default f-score will be set to 0.0, and `UndefinedMetricWarning` will be raised. This behavior can be modified by setting the `zero_division` parameter.

References

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import f1_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> f1_score(y_true, y_pred, average='macro')
0.26...
>>> f1_score(y_true, y_pred, average='micro')
0.33...
>>> f1_score(y_true, y_pred, average='weighted')
0.26...
>>> f1_score(y_true, y_pred, average=None)
array([0.8, 0. , 0. ])
```

```
>>> # binary classification
>>> y_true_empty = [0, 0, 0, 0, 0, 0]
>>> y_pred_empty = [0, 0, 0, 0, 0, 0]
>>> f1_score(y_true_empty, y_pred_empty)
0.0...
>>> f1_score(y_true_empty, y_pred_empty, zero_division=1.0)
1.0...
>>> f1_score(y_true_empty, y_pred_empty, zero_division=np.nan)
nan...
```

```
>>> # multilabel classification
>>> y_true = [[0, 0, 0], [1, 1, 1], [0, 1, 1]]
>>> y_pred = [[0, 0, 0], [1, 1, 1], [1, 1, 0]]
>>> f1_score(y_true, y_pred, average=None)
array([0.66666667, 1.        , 0.66666667])
```

roc_auc_score(*y_true*, *y_score*, ***, *average='macro'*, *sample_weight=None*, *max_fpr=None*, *multi_class='raise'*, *labels=None*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Note: this implementation can be used with binary, multiclass and multilabel classification, but some restrictions apply (see Parameters).

Read more in the User Guide.

Parameters

- `y_true` (*array-like of shape (n_samples,) or (n_samples, n_classes)*) – True labels or binary label indicators. The binary and multiclass cases expect labels with shape (n_samples,) while the multilabel case expects binary label indicators with shape (n_samples, n_classes).
- `y_score` (*array-like of shape (n_samples,) or (n_samples, n_classes)*) – Target scores.
  - In the binary case, it corresponds to an array of shape *(n_samples,)*. Both probability estimates and non-thresholded decision values can be provided. The probability estimates correspond to the **probability of the class with the greater label**, i.e. *estimator.classes_[1]* and thus *estimator.predict_proba(X, y)[:, 1]*. The decision values corresponds to the output of *estimator.decision_function(X, y)*. See more information in the User guide;
  - In the multiclass case, it corresponds to an array of shape *(n_samples, n_classes)* of probability estimates provided by the *predict_proba* method. The probability estimates **must** sum to 1 across the possible classes. In addition, the order of the class scores must correspond to the order of `labels`, if provided, or else to the numerical or lexicographical order of the labels in `y_true`. See more information in the User guide;
  - In the multilabel case, it corresponds to an array of shape *(n_samples, n_classes)*. Probability estimates are provided by the *predict_proba* method and the non-thresholded decision values by the *decision_function* method. The probability estimates correspond to the **probability of the class with the greater label for each output** of the classifier. See more information in the User guide.
- `average` (*{'micro', 'macro', 'samples', 'weighted'} or None, default='macro'*) – If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data. Note: multiclass ROC AUC currently only handles the 'macro' and 'weighted' averages. For multiclass targets, *average=None* is only implemented for *multi_class='ovr'* and *average='micro'* is only implemented for *multi_class='ovr'*.

  `'micro'`:
  Calculate metrics globally by considering each element of the label indicator matrix as a label.

  `'macro'`:
  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  `'weighted'`:
  Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

  `'samples'`:
  Calculate metrics for each instance, and find their average.

  Will be ignored when `y_true` is binary.
- `sample_weight` (*array-like of shape (n_samples,), default=None*) – Sample weights.
- `max_fpr` (*float > 0 and <= 1, default=None*) – If not `None`, the standardized partial AUC [2]_ over the range [0, max_fpr] is returned. For the multiclass case, `max_fpr`, should be either equal to `None` or `1.0` as AUC ROC partial computation currently is not supported for multiclass.

- **multi_class** (*{'raise', 'ovr', 'ovo'}, default='raise'*) – Only used for multiclass targets. Determines the type of configuration to use. The default value raises an error, so either `'ovr'` or `'ovo'` must be passed explicitly.

  'ovr':
  Stands for One-vs-rest. Computes the AUC of each class against the rest [3]_ [4]_. This treats the multiclass case in the same way as the multilabel case. Sensitive to class imbalance even when `average == 'macro'`, because class imbalance affects the composition of each of the 'rest' groupings.

  'ovo':
  Stands for One-vs-one. Computes the average AUC of all possible pairwise combinations of classes[5]. Insensitive to class imbalance when `average == 'macro'`.

- **labels** (*array-like of shape (n_classes,), default=None*) – Only used for multiclass targets. List of labels that index the classes in `y_score`. If `None`, the numerical or lexicographical order of the labels in `y_true` is used.

Returns
auc – Area Under the Curve score.

Return type
float

See also:

average_precision_score
Area under the precision-recall curve.

roc_curve
Compute Receiver operating characteristic (ROC) curve.

RocCurveDisplay.from_estimator
Plot Receiver Operating Characteristic (ROC) curve given an estimator and some data.

RocCurveDisplay.from_predictions
Plot Receiver Operating Characteristic (ROC) curve given the true and predicted values.

Notes

The Gini Coefficient is a summary measure of the ranking ability of binary classifiers. It is expressed using the area under of the ROC as follows:

G = 2 * AUC - 1

Where G is the Gini coefficient and AUC is the ROC-AUC score. This normalisation will ensure that random guessing will yield a score of 0 in expectation, and it is upper bounded by 1.

[5] Hand, D.J., Till, R.J. (2001). A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning, 45(2), 171-186.

References

Examples

Binary case:

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.metrics import roc_auc_score
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = LogisticRegression(solver="liblinear", random_state=0).fit(X, y)
>>> roc_auc_score(y, clf.predict_proba(X)[:, 1])
0.99...
>>> roc_auc_score(y, clf.decision_function(X))
0.99...
```

Multiclass case:

```
>>> from sklearn.datasets import load_iris
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(solver="liblinear").fit(X, y)
>>> roc_auc_score(y, clf.predict_proba(X), multi_class='ovr')
0.99...
```

Multilabel case:

```
>>> import numpy as np
>>> from sklearn.datasets import make_multilabel_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> X, y = make_multilabel_classification(random_state=0)
>>> clf = MultiOutputClassifier(clf).fit(X, y)
>>> # get a list of n_output containing probability arrays of shape
>>> # (n_samples, n_classes)
>>> y_pred = clf.predict_proba(X)
>>> # extract the positive columns for each output
>>> y_pred = np.transpose([pred[:, 1] for pred in y_pred])
>>> roc_auc_score(y, y_pred, average=None)
array([0.82..., 0.86..., 0.94..., 0.85... , 0.94...])
>>> from sklearn.linear_model import RidgeClassifierCV
>>> clf = RidgeClassifierCV().fit(X, y)
>>> roc_auc_score(y, clf.decision_function(X), average=None)
array([0.81..., 0.84... , 0.93..., 0.87..., 0.94...])
```

accuracy_score(*y_true*, *y_pred*, ***, *normalize=True*, *sample_weight=None*)

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) labels.

- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted labels, as returned by a classifier.
- **normalize** (*bool, default=True*) – If `False`, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

score – If `normalize == True`, return the fraction of correctly classified samples (float), else returns the number of correctly classified samples (int).

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

Return type

float

See also:

balanced_accuracy_score
Compute the balanced accuracy to deal with imbalanced datasets.

jaccard_score
Compute the Jaccard similarity coefficient score.

hamming_loss
Compute the average Hamming loss or Hamming distance between two sets of samples.

zero_one_loss
Compute the Zero-one classification loss. By default, the function will return the percentage of imperfectly predicted subsets.

Notes

In binary classification, this function is equal to the *jaccard_score* function.

Examples

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2.0
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

`balanced_accuracy_score`(*y_true*, *y_pred*, ***, *sample_weight=None*, *adjusted=False*)

Compute the balanced accuracy.

The balanced accuracy in binary and multiclass classification problems to deal with imbalanced datasets. It is defined as the average of recall obtained on each class.

The best value is 1 and the worst value is 0 when `adjusted=False`.

Read more in the User Guide.

New in version 0.20.

Parameters

- **`y_true`** (*`array-like of shape (n_samples,)`*) – Ground truth (correct) target values.
- **`y_pred`** (*`array-like of shape (n_samples,)`*) – Estimated targets as returned by a classifier.
- **`sample_weight`** (*`array-like of shape (n_samples,), default=None`*) – Sample weights.
- **`adjusted`** (*`bool, default=False`*) – When true, the result is adjusted for chance, so that random performance would score 0, while keeping perfect performance at a score of 1.

Returns

balanced_accuracy – Balanced accuracy score.

Return type

float

See also:

`average_precision_score`
Compute average precision (AP) from prediction scores.

`precision_score`
Compute the precision score.

`recall_score`
Compute the recall score.

`roc_auc_score`
Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Notes

Some literature promotes alternative definitions of balanced accuracy. Our definition is equivalent to *`accuracy_score()`* with class-balanced sample weights, and shares desirable properties with the binary case. See the User Guide.

References

Examples

```
>>> from sklearn.metrics import balanced_accuracy_score
>>> y_true = [0, 1, 0, 0, 1, 0]
>>> y_pred = [0, 1, 0, 0, 0, 1]
>>> balanced_accuracy_score(y_true, y_pred)
0.625
```

top_k_accuracy_score(*y_true*, *y_score*, ***, *k=2*, *normalize=True*, *sample_weight=None*, *labels=None*)

Top-k Accuracy classification score.

This metric computes the number of times where the correct label is among the top *k* labels predicted (ranked by predicted scores). Note that the multilabel case isn't covered here.

Read more in the User Guide

Parameters

- **y_true** (*array-like of shape (n_samples,)*) – True labels.
- **y_score** (*array-like of shape (n_samples,) or (n_samples, n_classes)*) – Target scores. These can be either probability estimates or non-thresholded decision values (as returned by decision_function on some classifiers). The binary case expects scores with shape (n_samples,) while the multiclass case expects scores with shape (n_samples, n_classes). In the multiclass case, the order of the class scores must correspond to the order of `labels`, if provided, or else to the numerical or lexicographical order of the labels in `y_true`. If `y_true` does not contain all the labels, `labels` must be provided.
- **k** (*int, default=2*) – Number of most likely outcomes considered to find the correct label.
- **normalize** (*bool, default=True*) – If *True*, return the fraction of correctly classified samples. Otherwise, return the number of correctly classified samples.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights. If *None*, all samples are given the same weight.
- **labels** (*array-like of shape (n_classes,), default=None*) – Multiclass only. List of labels that index the classes in `y_score`. If `None`, the numerical or lexicographical order of the labels in `y_true` is used. If `y_true` does not contain all the labels, `labels` must be provided.

Returns

score – The top-k accuracy score. The best performance is 1 with *normalize == True* and the number of samples with *normalize == False*.

Return type

float

See also:

accuracy_score
Compute the accuracy score. By default, the function will return the fraction of correct predictions divided by the total number of predictions.

Notes

In cases where two or more labels are assigned equal predicted scores, the labels with the highest indices will be chosen first. This might impact the result if the correct label falls after the threshold because of that.

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import top_k_accuracy_score
>>> y_true = np.array([0, 1, 2, 2])
>>> y_score = np.array([[0.5, 0.2, 0.2],  # 0 is in top 2
...                     [0.3, 0.4, 0.2],  # 1 is in top 2
...                     [0.2, 0.4, 0.3],  # 2 is in top 2
...                     [0.7, 0.2, 0.1]]) # 2 isn't in top 2
>>> top_k_accuracy_score(y_true, y_score, k=2)
0.75
>>> # Not normalizing gives the number of "correctly" classified samples
>>> top_k_accuracy_score(y_true, y_score, k=2, normalize=False)
3
```

`pearson_r2_score`(*y: ndarray*, *y_pred: ndarray*) → float

Computes Pearson R^2 (square of Pearson correlation).

Parameters

- **y** (*np.ndarray*) – ground truth array
- **y_pred** (*np.ndarray*) – predicted array

Returns
The Pearson-R^2 score.

Return type
float

`jaccard_index`(*y: ndarray*, *y_pred: ndarray*) → float

Computes Jaccard Index which is the Intersection Over Union metric which is commonly used in image segmentation tasks.

DEPRECATED: WILL BE REMOVED IN A FUTURE VERSION OF DEEEPCHEM. USE *jaccard_score* instead.

Parameters

- **y** (*np.ndarray*) – ground truth array
- **y_pred** (*np.ndarray*) – predicted array

Returns
score – The jaccard index. A number between 0 and 1.

Return type
float

`pixel_error`(*y: ndarray*, *y_pred: ndarray*) → float

An error metric in case y, y_pred are images.

Defined as 1 - the maximal F-score of pixel similarity, or squared Euclidean distance between the original and the result labels.

Parameters

- **y** (*np.ndarray*) – ground truth array
- **y_pred** (*np.ndarray*) – predicted array

Returns
score – The pixel-error. A number between 0 and 1.

Return type
float

prc_auc_score(*y: ndarray*, *y_pred: ndarray*) $\rightarrow$ float

Compute area under precision-recall curve

Parameters

- **y** (*np.ndarray*) – A numpy array of shape *(N, n_classes)* or *(N,)* with true labels
- **y_pred** (*np.ndarray*) – Of shape *(N, n_classes)* with class probabilities.

Returns
The area under the precision-recall curve. A number between 0 and 1.

Return type
float

rms_score(*y_true: ndarray*, *y_pred: ndarray*) $\rightarrow$ float

Computes RMS error.

mae_score(*y_true: ndarray*, *y_pred: ndarray*) $\rightarrow$ float

Computes MAE.

kappa_score(*y1*, *y2*, ***, *labels=None*, *weights=None*, *sample_weight=None*)

Compute Cohen's kappa: a statistic that measures inter-annotator agreement.

This function computes Cohen's kappa [1]_, a score that expresses the level of agreement between two annotators on a classification problem. It is defined as

$$\kappa = (p_o - p_e)/(1 - p_e)$$

where p_o is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio), and p_e is the expected agreement when both annotators assign labels randomly. p_e is estimated using a per-annotator empirical prior over the class labels [2]_.

Read more in the User Guide.

Parameters

- **y1** (*array-like of shape (n_samples,)*) – Labels assigned by the first annotator.
- **y2** (*array-like of shape (n_samples,)*) – Labels assigned by the second annotator. The kappa statistic is symmetric, so swapping `y1` and `y2` doesn't change the value.
- **labels** (*array-like of shape (n_classes,), default=None*) – List of labels to index the matrix. This may be used to select a subset of labels. If *None*, all labels that appear at least once in `y1` or `y2` are used.
- **weights** (*{'linear', 'quadratic'}, default=None*) – Weighting type to calculate the score. *None* means no weighted; "linear" means linear weighted; "quadratic" means quadratic weighted.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns
kappa – The kappa statistic, which is a number between -1 and 1. The maximum value means complete agreement; zero or lower means chance agreement.

Return type
float

References

Examples

```
>>> from sklearn.metrics import cohen_kappa_score
>>> y1 = ["negative", "positive", "negative", "neutral", "positive"]
>>> y2 = ["negative", "positive", "negative", "neutral", "negative"]
>>> cohen_kappa_score(y1, y2)
0.6875
```

bedroc_score(*y_true: ndarray*, *y_pred: ndarray*, *alpha: float = 20.0*)

Compute BEDROC metric.

BEDROC metric implemented according to Truchon and Bayley that modifies the ROC score by allowing for a factor of early recognition. Please confirm details from [1]_.

Parameters

- **y_true** (*np.ndarray*) – Binary class labels. 1 for positive class, 0 otherwise
- **y_pred** (*np.ndarray*) – Predicted labels
- **alpha** (*float, default 20.0*) – Early recognition parameter

Returns
Value in [0, 1] that indicates the degree of early recognition

Return type
float

Notes

This function requires RDKit to be installed.

References

concordance_index(*y_true: ndarray*, *y_pred: ndarray*) → float

Compute Concordance index.

Statistical metric indicates the quality of the predicted ranking. Please confirm details from [1]_.

Parameters

- **y_true** (*np.ndarray*) – continous value
- **y_pred** (*np.ndarray*) – Predicted value

Returns
score between [0,1]

Return type
float

References

`get_motif_scores`(*encoded_sequences: ndarray*, *motif_names: List[str]*, *max_scores: int | None = None*, *return_positions: bool = False*, *GC_fraction: float = 0.4*) → ndarray

Computes pwm log odds.

Parameters

- **encoded_sequences** (*np.ndarray*) – A numpy array of shape *(N_sequences, N_letters, sequence_length, 1)*.
- **motif_names** (*List[str]*) – List of motif file names.
- **max_scores** (*int, optional*) – Get top *max_scores* scores.
- **return_positions** (*bool, default False*) – Whether to return postions or not.
- **GC_fraction** (*float, default 0.4*) – GC fraction in background sequence.

Returns
A numpy array of complete score. The shape is *(N_sequences, num_motifs, seq_length)* by default. If max_scores, the shape of score array is *(N_sequences, num_motifs*max_scores)*. If max_scores and return_positions, the shape of score array with max scores and their positions. is *(N_sequences, 2*num_motifs*max_scores)*.

Return type
np.ndarray

Notes

This method requires simdna to be installed.

`get_pssm_scores`(*encoded_sequences: ndarray*, *pssm: ndarray*) → ndarray

Convolves pssm and its reverse complement with encoded sequences and returns the maximum score at each position of each sequence.

Parameters

- **encoded_sequences** (*np.ndarray*) – A numpy array of shape *(N_sequences, N_letters, sequence_length, 1)*.
- **pssm** (*np.ndarray*) – A numpy array of shape *(4, pssm_length)*.

Returns
scores – A numpy array of shape *(N_sequences, sequence_length)*.

Return type
np.ndarray

`in_silico_mutagenesis`(*model: Model*, *encoded_sequences: ndarray*) → ndarray

Computes in-silico-mutagenesis scores

Parameters

- **model** (*Model*) – This can be any model that accepts inputs of the required shape and produces an output of shape *(N_sequences, N_tasks)*.

- **encoded_sequences** (`np.ndarray`) – A numpy array of shape *(N_sequences, N_letters, sequence_length, 1)*

Returns
A numpy array of ISM scores. The shape is *(num_task, N_sequences, N_letters, sequence_length, 1)*.

Return type
np.ndarray

3.24.4 Metric Class

The `dc.metrics.Metric` class is a wrapper around metric functions which interoperates with DeepChem `dc.models.Model`.

class Metric(*metric: Callable[[...], float]*, *task_averager: Callable[[...], Any] | None = None*, *name: str | None = None*, *threshold: float | None = None*, *mode: str | None = None*, *n_tasks: int | None = None*, *classification_handling_mode: str | None = None*, *threshold_value: float | None = None*)

Wrapper class for computing user-defined metrics.

The *Metric* class provides a wrapper for standardizing the API around different classes of metrics that may be useful for DeepChem models. The implementation provides a few non-standard conveniences such as built-in support for multitask and multiclass metrics.

There are a variety of different metrics this class aims to support. Metrics for classification and regression that assume that values to compare are scalars are supported.

At present, this class doesn't support metric computation on models which don't present scalar outputs. For example, if you have a generative model which predicts images or molecules, you will need to write a custom evaluation and metric setup.

__init__(*metric: Callable[[...], float]*, *task_averager: Callable[[...], Any] | None = None*, *name: str | None = None*, *threshold: float | None = None*, *mode: str | None = None*, *n_tasks: int | None = None*, *classification_handling_mode: str | None = None*, *threshold_value: float | None = None*)

Parameters

- **metric** (`function`) – Function that takes args y_true, y_pred (in that order) and computes desired score. If sample weights are to be considered, *metric* may take in an additional keyword argument *sample_weight*.
- **task_averager** (`function, default None`) – If not None, should be a function that averages metrics across tasks.
- **name** (`str, default None`) – Name of this metric
- **threshold** (`float, default None (DEPRECATED)`) – Used for binary metrics and is the threshold for the positive class.
- **mode** (`str, default None`) – Should usually be "classification" or "regression."
- **n_tasks** (`int, default None`) – The number of tasks this class is expected to handle.
- **classification_handling_mode** (`str, default None`) – DeepChem models by default predict class probabilities for classification problems. This means that for a given singletask prediction, after shape normalization, the DeepChem labels and prediction will be numpy arrays of shape *(n_samples, n_tasks, n_classes)* with class probabilities. *classification_handling_mode* is a string that instructs this method how to handle transforming these probabilities. It can take on the following values: - "direct": Pass *y_true* and *y_pred*

directy into *self.metric*. - "threshold": Use *threshold_predictions* to threshold *y_true* and *y_pred*.

Use *threshold_value* as the desired threshold. This converts them into arrays of shape *(n_samples, n_tasks)*, where each element is a class index.

- "threshold-one-hot": Use *threshold_predictions* to threshold *y_true* and *y_pred* using *threshold_values*, then apply *to_one_hot* to output.
- None: Select a mode automatically based on the metric.

- **threshold_value** (*float, default None*) – If set, and *classification_handling_mode* is "threshold" or "threshold-one-hot", apply a thresholding operation to values with this threshold. This option is only sensible on binary classification tasks. For multiclass problems, or if *threshold_value* is None, argmax() is used to select the highest probability class for each task.

compute_metric(*y_true: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y_pred: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *n_tasks: int | None = None*, *n_classes: int = 2*, *per_task_metrics: bool = False*, *use_sample_weights: bool = False*, ***kwargs*) → Any

Compute a performance metric for each task.

Parameters

- **y_true** (*ArrayLike*) – An ArrayLike containing true values for each task. Must be of shape *(N,)* or *(N, n_tasks)* or *(N, n_tasks, n_classes)* if a classification metric. If of shape *(N, n_tasks)* values can either be class-labels or probabilities of the positive class for binary classification problems. If a regression problem, must be of shape *(N,)* or *(N, n_tasks)* or *(N, n_tasks, 1)* if a regression metric.
- **y_pred** (*ArrayLike*) – An ArrayLike containing predicted values for each task. Must be of shape *(N, n_tasks, n_classes)* if a classification metric, else must be of shape *(N, n_tasks)* if a regression metric.
- **w** (*ArrayLike, default None*) – An ArrayLike containing weights for each datapoint. If specified, must be of shape *(N, n_tasks)*.
- **n_tasks** (*int, default None*) – The number of tasks this class is expected to handle.
- **n_classes** (*int, default 2*) – Number of classes in data for classification tasks.
- **per_task_metrics** (*bool, default False*) – If true, return computed metric for each task on multitask dataset.
- **use_sample_weights** (*bool, default False*) – If set, use per-sample weights *w*.
- **kwargs** (*dict*) – Will be passed on to self.metric

Returns

A numpy array containing metric values for each task.

Return type

np.ndarray

`compute_singletask_metric`(*y_true: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *y_pred: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *w: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *n_samples: int | None = None*, *use_sample_weights: bool = False*, ***kwargs*) → float

Compute a metric value.

Parameters

- **`y_true`** (*`ArrayLike`*) – True values array. This array must be of shape *(N, n_classes)* if classification and *(N,)* if regression.
- **`y_pred`** (*`ArrayLike`*) – Predictions array. This array must be of shape *(N, n_classes)* if classification and *(N,)* if regression.
- **`w`** (*`ArrayLike, default None`*) – Sample weight array. This array must be of shape *(N,)*
- **`n_samples`** (*`int, default None (DEPRECATED)`*) – The number of samples in the dataset. This is *N*. This argument is ignored.
- **`use_sample_weights`** (*`bool, default False`*) – If set, use per-sample weights *w*.
- **`kwargs`** (*`dict`*) – Will be passed on to self.metric

Returns

metric_value – The computed value of the metric.

Return type

float

3.25 Hyperparameter Tuning

One of the most important aspects of machine learning is hyperparameter tuning. Many machine learning models have a number of hyperparameters that control aspects of the model. These hyperparameters typically cannot be learned directly by the same learning algorithm used for the rest of learning and have to be set in an alternate fashion. The `dc.hyper` module contains utilities for hyperparameter tuning.

DeepChem's hyperparameter optimzation algorithms are simple and run in single-threaded fashion. They are not intended to be production grade hyperparameter utilities, but rather useful first tools as you start exploring your parameter space. As the needs of your application grow, we recommend swapping to a more heavy duty hyperparameter optimization library.

3.25.1 Hyperparameter Optimization API

`class HyperparamOpt`(*model_builder: Callable[[...],* Model*]*)

Abstract superclass for hyperparameter search classes.

This class is an abstract base class for hyperparameter search classes in DeepChem. Hyperparameter search is performed on *dc.models.Model* classes. Each hyperparameter object accepts a *dc.models.Model* class upon construct. When the *hyperparam_search* class is invoked, this class is used to construct many different concrete models which are trained on the specified training set and evaluated on a given validation set.

Different subclasses of *HyperparamOpt* differ in the choice of strategy for searching the hyperparameter evaluation space. This class itself is an abstract superclass and should never be directly instantiated.

`__init__`(*model_builder: Callable[[...],* Model*]*)

Initialize Hyperparameter Optimizer.

Note this is an abstract constructor which should only be used by subclasses.

Parameters

model_builder (`constructor function.`) – This parameter must be constructor function which returns an object which is an instance of *dc.models.Model*. This function must accept two arguments, *model_params* of type *dict* and *model_dir*, a string specifying a path to a model directory. See the example.

`hyperparam_search`(*params_dict: Dict*, *train_dataset:* Dataset, *valid_dataset:* Dataset, *metric:* Metric, *output_transformers: List[*Transformer*] = []*, *nb_epoch: int = 10*, *use_max: bool = True*, *logfile: str = 'results.txt'*, *logdir: str | None = None*, ***kwargs*) → Tuple[*Model*, Dict[str, Any], Dict[str, Any]]

Conduct Hyperparameter search.

This method defines the common API shared by all hyperparameter optimization subclasses. Different classes will implement different search methods but they must all follow this common API.

Parameters

- **params_dict** (`Dict`) – Dictionary mapping strings to values. Note that the precise semantics of *params_dict* will change depending on the optimizer that you're using. Depending on the type of hyperparameter optimization, these values can be ints/floats/strings/lists/etc. Read the documentation for the concrete hyperparameter optimization subclass you're using to learn more about what's expected.
- **train_dataset** (`Dataset`) – dataset used for training
- **valid_dataset** (`Dataset`) – dataset used for validation(optimization on valid scores)
- **metric** (`Metric`) – metric used for evaluation
- **output_transformers** (`list[Transformer]`) – Transformers for evaluation. This argument is needed since *train_dataset* and *valid_dataset* may have been transformed for learning and need the transform to be inverted before the metric can be evaluated on a model.
- **nb_epoch** (`int, (default 10)`) – Specifies the number of training epochs during each iteration of optimization.
- **use_max** (`bool, optional`) – If True, return the model with the highest score. Else return model with the minimum score.
- **logdir** (`str, optional`) – The directory in which to store created models. If not set, will use a temporary directory.

- **logfile** (str, optional (default *results.txt*)) – Name of logfile to write results to. If specified, this must be a valid file name. If not specified, results of hyperparameter search will be written to *logdir/results.txt*.

Returns
(best_model, best_hyperparams, all_scores) where *best_model* is an instance of *dc.models.Model*, *best_hyperparams* is a dictionary of parameters, and *all_scores* is a dictionary mapping string representations of hyperparameter sets to validation scores.

Return type
Tuple[*best_model*, *best_hyperparams*, *all_scores*]

3.25.2 Grid Hyperparameter Optimization

This is the simplest form of hyperparameter optimization that simply involves iterating over a fixed grid of possible values for hyperaparameters.

class GridHyperparamOpt(*model_builder: Callable[[...], Model]*)

Provides simple grid hyperparameter search capabilities.

This class performs a grid hyperparameter search over the specified hyperparameter space. This implementation is simple and simply does a direct iteration over all possible hyperparameters and doesn't use parallelization to speed up the search.

Examples

This example shows the type of constructor function expected.

```
>>> import sklearn
>>> import deepchem as dc
>>> optimizer = dc.hyper.GridHyperparamOpt(lambda **p: dc.models.
↪GraphConvModel(**p))
```

Here's a more sophisticated example that shows how to optimize only some parameters of a model. In this case, we have some parameters we want to optimize, and others which we don't. To handle this type of search, we create a *model_builder* which hard codes some arguments (in this case, *max_iter* is a hyperparameter which we don't want to search over)

```
>>> import deepchem as dc
>>> import numpy as np
>>> from sklearn.linear_model import LogisticRegression as LR
>>> # generating data
>>> X = np.arange(1, 11, 1).reshape(-1, 1)
>>> y = np.hstack((np.zeros(5), np.ones(5)))
>>> dataset = dc.data.NumpyDataset(X, y)
>>> # splitting dataset into train and test
>>> splitter = dc.splits.RandomSplitter()
>>> train_dataset, test_dataset = splitter.train_test_split(dataset)
>>> # metric to evaluate result of a set of parameters
>>> metric = dc.metrics.Metric(dc.metrics.accuracy_score)
>>> # defining `model_builder`
>>> def model_builder(**model_params):
...   penalty = model_params['penalty']
...   solver = model_params['solver']
```

(continues on next page)

(continued from previous page)

```
...   lr = LR(penalty=penalty, solver=solver, max_iter=100)
...   return dc.models.SklearnModel(lr)
>>> # the parameters which are to be optimized
>>> params = {
...   'penalty': ['l1', 'l2'],
...   'solver': ['liblinear', 'saga']
...   }
>>> # Creating optimizer and searching over hyperparameters
>>> optimizer = dc.hyper.GridHyperparamOpt(model_builder)
>>> best_model, best_hyperparams, all_results =     optimizer.hyperparam_
↪search(params, train_dataset, test_dataset, metric)
>>> best_hyperparams  # the best hyperparameters
{'penalty': 'l2', 'solver': 'saga'}
```

hyperparam_search(*params_dict: Dict*, *train_dataset:* Dataset, *valid_dataset:* Dataset, *metric:* Metric, *output_transformers: List[*Transformer*] = []*, *nb_epoch: int = 10*, *use_max: bool = True*, *logfile: str = 'results.txt'*, *logdir: str | None = None*, ***kwargs*) → Tuple[*Model*, Dict, Dict]

Perform hyperparams search according to params_dict.

Each key to hyperparams_dict is a model_param. The values should be a list of potential values for that hyperparam.

Parameters

- **params_dict** (*Dict*) – Maps hyperparameter names (strings) to lists of possible parameter values.
- **train_dataset** (Dataset) – dataset used for training
- **valid_dataset** (Dataset) – dataset used for validation(optimization on valid scores)
- **metric** (Metric) – metric used for evaluation
- **output_transformers** (*list[*Transformer*]*) – Transformers for evaluation. This argument is needed since *train_dataset* and *valid_dataset* may have been transformed for learning and need the transform to be inverted before the metric can be evaluated on a model.
- **nb_epoch** (*int, (default 10)*) – Specifies the number of training epochs during each iteration of optimization. Not used by all model types.
- **use_max** (*bool, optional*) – If True, return the model with the highest score. Else return model with the minimum score.
- **logdir** (*str, optional*) – The directory in which to store created models. If not set, will use a temporary directory.
- **logfile** (str, optional (default *results.txt*)) – Name of logfile to write results to. If specified, this is must be a valid file name. If not specified, results of hyperparameter search will be written to *logdir/results.txt*.

Returns

- Tuple[*best_model*, *best_hyperparams*, *all_scores*]
- *(best_model, best_hyperparams, all_scores)* where *best_model* is
- an instance of *dc.model.Model*, *best_hyperparams* is a

- dictionary of parameters, and *all_scores* is a dictionary mapping
- *string representations of hyperparameter sets to validation*
- *scores.*

Notes

From DeepChem 2.6, the return type of *best_hyperparams* is a dictionary of parameters rather than a tuple of parameters as it was previously. The new changes have been made to standardize the behaviour across different hyperparameter optimization techniques available in DeepChem.

3.25.3 Gaussian Process Hyperparameter Optimization

`class GaussianProcessHyperparamOpt`(*model_builder: Callable[[...], Model], max_iter: int = 20, search_range: int | float | Dict = 4*)

Gaussian Process Global Optimization(GPGO)

This class uses Gaussian Process optimization to select hyperparameters. Underneath the hood it uses pyGPGO to optimize models. If you don't have pyGPGO installed, you won't be able to use this class.

Note that *params_dict* has a different semantics than for *GridHyperparamOpt*. *param_dict[hp]* must be an int/float and is used as the center of a search range.

Examples

This example shows the type of constructor function expected.

```
>>> import deepchem as dc
>>> optimizer = dc.hyper.GaussianProcessHyperparamOpt(lambda **p: dc.models.
↪GraphConvModel(n_tasks=1, **p))
```

Here's a more sophisticated example that shows how to optimize only some parameters of a model. In this case, we have some parameters we want to optimize, and others which we don't. To handle this type of search, we create a *model_builder* which hard codes some arguments (in this case, *n_tasks* and *n_features* which are properties of a dataset and not hyperparameters to search over.)

```
>>> import numpy as np
>>> from sklearn.ensemble import RandomForestRegressor as RF
>>> def model_builder(**model_params):
...   n_estimators = model_params['n_estimators']
...   min_samples_split = model_params['min_samples_split']
...   rf_model = RF(n_estimators=n_estimators, min_samples_split=min_samples_split)
...   rf_model = RF(n_estimators=n_estimators)
...   return dc.models.SklearnModel(rf_model)
>>> optimizer = dc.hyper.GaussianProcessHyperparamOpt(model_builder)
>>> params_dict = {"n_estimators":100, "min_samples_split":2}
>>> train_dataset = dc.data.NumpyDataset(X=np.random.rand(50, 5),
...   y=np.random.rand(50, 1))
>>> valid_dataset = dc.data.NumpyDataset(X=np.random.rand(20, 5),
...   y=np.random.rand(20, 1))
>>> metric = dc.metrics.Metric(dc.metrics.pearson_r2_score)
```

>> best_model, best_hyperparams, all_results = optimizer.hyperparam_search(params_dict, train_dataset, valid_dataset, metric, max_iter=2) >> type(best_hyperparams) <class 'dict'>

Parameters

- **`model_builder`** (*`constructor function.`*) – This parameter must be constructor function which returns an object which is an instance of *dc.models.Model*. This function must accept two arguments, *model_params* of type *dict* and *model_dir*, a string specifying a path to a model directory.
- **`max_iter`** (*`int, default 20`*) – number of optimization trials
- **`search_range`** (*`int/float/Dict (default 4)`*) – The *search_range* specifies the range of parameter values to search for. If *search_range* is an int/float, it is used as the global search range for parameters. This creates a search problem on the following space:

  optimization on [initial value / search_range,
  initial value * search_range]

  If *search_range* is a dict, it must contain the same keys as for *params_dict*. In this case, *search_range* specifies a per-parameter search range. This is useful in case some parameters have a larger natural range than others. For a given hyperparameter *hp* this would create the following search range:

  optimization on hp on [initial value[hp] / search_range[hp],
  initial value[hp] * search_range[hp]]

Notes

This class requires pyGPGO to be installed.

`__init__`(*model_builder: Callable[[...], Model], max_iter: int = 20, search_range: int | float | Dict = 4*)

Initialize Hyperparameter Optimizer.

Note this is an abstract constructor which should only be used by subclasses.

Parameters

`model_builder` (*`constructor function.`*) – This parameter must be constructor function which returns an object which is an instance of *dc.models.Model*. This function must accept two arguments, *model_params* of type *dict* and *model_dir*, a string specifying a path to a model directory. See the example.

`hyperparam_search`(*params_dict: Dict, train_dataset: Dataset, valid_dataset: Dataset, metric: Metric, output_transformers: List[Transformer] = [], nb_epoch: int = 10, use_max: bool = True, logfile: str = 'results.txt', logdir: str | None = None, **kwargs*) → Tuple[*Model*, Dict[str, Any], Dict[str, Any]]

Perform hyperparameter search using a gaussian process.

Parameters

- **`params_dict`** (*`Dict`*) – Maps hyperparameter names (strings) to possible parameter values. The semantics of this list are different than for *GridHyperparamOpt*. *params_dict[hp]* must map to an int/float, which is used as the center of a search with radius *search_range* since pyGPGO can only optimize numerical hyperparameters.
- **`train_dataset`** (`Dataset`) – dataset used for training
- **`valid_dataset`** (`Dataset`) – dataset used for validation(optimization on valid scores)
- **`metric`** (`Metric`) – metric used for evaluation

- **output_transformers** (*list[Transformer]*) – Transformers for evaluation. This argument is needed since *train_dataset* and *valid_dataset* may have been transformed for learning and need the transform to be inverted before the metric can be evaluated on a model.
- **nb_epoch** (*int, (default 10)*) – Specifies the number of training epochs during each iteration of optimization. Not used by all model types.
- **use_max** (*bool, (default True)*) – Specifies whether to maximize or minimize *metric*. maximization(True) or minimization(False)
- **logdir** (*str, optional, (default None)*) – The directory in which to store created models. If not set, will use a temporary directory.
- **logfile** (str, optional (default *results.txt*)) – Name of logfile to write results to. If specified, this is must be a valid file. If not specified, results of hyperparameter search will be written to *logdir/results.txt*.

Returns

(best_model, best_hyperparams, all_scores) where *best_model* is an instance of *dc.model.Model*, *best_hyperparams* is a dictionary of parameters, and *all_scores* is a dictionary mapping string representations of hyperparameter sets to validation scores.

Return type

Tuple[*best_model*, *best_hyperparams*, *all_scores*]

3.26 Metalearning

One of the hardest challenges in scientific machine learning is lack of access of sufficient data. Sometimes experiments are slow and expensive and there's no easy way to gain access to more data. What do you do then?

This module contains a collection of techniques for doing low data learning. "Metalearning" traditionally refers to techniques for "learning to learn" but here we take it to mean any technique which proves effective for learning with low amounts of data.

3.26.1 MetaLearner

This is the abstract superclass for metalearning algorithms.

class MetaLearner

Model and data to which the MAML algorithm can be applied.

To use MAML, create a subclass of this defining the learning problem to solve. It consists of a model that can be trained to perform many different tasks, and data for training it on a large (possibly infinite) set of different tasks.

compute_model(*inputs*, *variables*, *training*)

Compute the model for a set of inputs and variables.

Parameters

- **inputs** (*list of tensors*) – the inputs to the model
- **variables** (*list of tensors*) – the values to use for the model's variables. This might be the actual variables (as returned by the MetaLearner's variables property), or alternatively it might be the values of those variables after one or more steps of gradient descent for the current task.

- **training** (*bool*) – indicates whether the model is being invoked for training or prediction

Returns

- *(loss, outputs) where loss is the value of the model's loss function, and*
- *outputs is a list of the model's outputs*

property variables

Get the list of variables to train.

select_task()

Select a new task to train on.

If there is a fixed set of training tasks, this will typically cycle through them. If there are infinitely many training tasks, this can simply select a new one each time it is called.

get_batch()

Get a batch of data for training.

This should return the data as a list of arrays, one for each of the model's inputs. This will usually be called twice for each task, and should return a different batch on each call.

parameters()

Returns an iterator over the MetaLearner parameters.

3.27 Tensorflow implementation

3.27.1 MAML

class MAML(*learner*, *learning_rate=0.001*, *optimization_steps=1*, *meta_batch_size=10*, *optimizer=<deepchem.models.optimizers.Adam object>*, *model_dir=None*)

Implements the Model-Agnostic Meta-Learning algorithm for low data learning.

The algorithm is described in Finn et al., "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks" (https://arxiv.org/abs/1703.03400). It is used for training models that can perform a variety of tasks, depending on what data they are trained on. It assumes you have training data for many tasks, but only a small amount for each one. It performs "meta-learning" by looping over tasks and trying to minimize the loss on each one *after* one or a few steps of gradient descent. That is, it does not try to create a model that can directly solve the tasks, but rather tries to create a model that is very easy to train.

To use this class, create a subclass of MetaLearner that encapsulates the model and data for your learning problem. Pass it to a MAML object and call fit(). You can then use train_on_current_task() to fine tune the model for a particular task.

__init__(*learner*, *learning_rate=0.001*, *optimization_steps=1*, *meta_batch_size=10*, *optimizer=<deepchem.models.optimizers.Adam object>*, *model_dir=None*)

Create an object for performing meta-optimization.

Parameters

- **learner** (MetaLearner) – defines the meta-learning problem
- **learning_rate** (*float or Tensor*) – the learning rate to use for optimizing each task (not to be confused with the one used for meta-learning). This can optionally be made a variable (represented as a Tensor), in which case the learning rate will itself be learnable.

- **optimization_steps** (*int*) – the number of steps of gradient descent to perform for each task
- **meta_batch_size** (*int*) – the number of tasks to use for each step of meta-learning
- **optimizer** (Optimizer) – the optimizer to use for meta-learning (not to be confused with the gradient descent optimization performed for each task)
- **model_dir** (*str*) – the directory in which the model will be saved. If None, a temporary directory will be created.

fit(*steps*, *max_checkpoints_to_keep=5*, *checkpoint_interval=600*, *restore=False*)

Perform meta-learning to train the model.

Parameters

- **steps** (*int*) – the number of steps of meta-learning to perform
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **checkpoint_interval** (*float*) – the time interval at which to save checkpoints, measured in seconds
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint before training it further

restore()

Reload the model parameters from the most recent checkpoint file.

train_on_current_task(*optimization_steps=1*, *restore=True*)

Perform a few steps of gradient descent to fine tune the model on the current task.

Parameters

- **optimization_steps** (*int*) – the number of steps of gradient descent to perform
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint before optimizing

predict_on_batch(*inputs*)

Compute the model's outputs for a batch of inputs.

Parameters
inputs (*list of arrays*) – the inputs to the model

Returns

- *(loss, outputs) where loss is the value of the model's loss function, and*
- *outputs is a list of the model's outputs*

3.28 Torch implementation

3.28.1 MAML

class MAML(*learner: ~deepchem.metalearning.MetaLearner*, *learning_rate: float | ~deepchem.models.optimizers.LearningRateSchedule = 0.001*, *optimization_steps: int = 1*, *meta_batch_size: int = 10*, *optimizer: ~deepchem.models.optimizers.Optimizer = <deepchem.models.optimizers.Adam object>*, *model_dir: str | None = None*, *device: ~torch.device | None = None*)

Implements the Model-Agnostic Meta-Learning algorithm for low data learning.

The algorithm is described in Finn et al., "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks" (https://arxiv.org/abs/1703.03400). It is used for training models that can perform a variety of tasks, depending on what data they are trained on. It assumes you have training data for many tasks, but only a small amount for each one. It performs "meta-learning" by looping over tasks and trying to minimize the loss on each one *after* one or a few steps of gradient descent. That is, it does not try to create a model that can directly solve the tasks, but rather tries to create a model that is very easy to train.

To use this class, create a subclass of MetaLearner that encapsulates the model and data for your learning problem. Pass it to a MAML object and call fit(). You can then use train_on_current_task() to fine tune the model for a particular task. .. rubric:: Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> from deepchem.metalearning.torch_maml import MetaLearner, MAML
>>> class SineLearner(MetaLearner):
...     def __init__(self):
...         self.batch_size = 10
...         self.w1 = torch.nn.Parameter(torch.tensor(np.random.normal(size=[1, 40],
↪ scale=1.0),requires_grad=True))
...         self.w2 = torch.nn.Parameter(torch.tensor(np.random.normal(size=[40,␣
↪40], scale=np.sqrt(1 / 40)),requires_grad=True))
...         self.w3 = torch.nn.Parameter(torch.tensor(np.random.normal(size=[40, 1],
↪ scale=np.sqrt(1 / 40)),requires_grad=True))
...         self.b1 = torch.nn.Parameter(torch.tensor(np.zeros(40)),requires_
↪grad=True)
...         self.b2 = torch.nn.Parameter(torch.tensor(np.zeros(40)),requires_
↪grad=True)
...         self.b3 = torch.nn.Parameter(torch.tensor(np.zeros(1)),requires_
↪grad=True)
...     def compute_model(self, inputs, variables, training):
...         x, y = inputs
...         w1, w2, w3, b1, b2, b3 = variables
...         dense1 = F.relu(torch.matmul(x, w1) + b1)
...         dense2 = F.relu(torch.matmul(dense1, w2) + b2)
...         output = torch.matmul(dense2, w3) + b3
...         loss = torch.mean(torch.square(output - y))
...         return loss, [output]
...     @property
...     def variables(self):
...         return [self.w1, self.w2, self.w3, self.b1, self.b2, self.b3]
...     def select_task(self):
...         self.amplitude = 5.0 * np.random.random()
...         self.phase = np.pi * np.random.random()
...     def get_batch(self):
...         x = torch.tensor(np.random.uniform(-5.0, 5.0, (self.batch_size, 1)))
...         return [x, torch.tensor(self.amplitude * np.sin(x + self.phase))]
...     def parameters(self):
...         for key, value in self.__dict__.items():
...             if isinstance(value, torch.nn.Parameter):
```

(continues on next page)

(continued from previous page)

```
...                 yield value
>>> learner = SineLearner()
>>> optimizer = dc.models.optimizers.Adam(learning_rate=5e-3)
>>> maml = MAML(learner,meta_batch_size=4,optimizer=optimizer)
>>> maml.fit(9000)
```

To test it out on a new task and see how it works

```
>>> learner.select_task()
>>> maml.restore()
>>> batch = learner.get_batch()
>>> loss, outputs = maml.predict_on_batch(batch)
>>> maml.train_on_current_task()
>>> loss, outputs = maml.predict_on_batch(batch)
```

`__init__`(*learner: ~deepchem.metalearning.MetaLearner*, *learning_rate: float | ~deepchem.models.optimizers.LearningRateSchedule = 0.001*, *optimization_steps: int = 1*, *meta_batch_size: int = 10*, *optimizer: ~deepchem.models.optimizers.Optimizer = <deepchem.models.optimizers.Adam object>*, *model_dir: str | None = None*, *device: ~torch.device | None = None*)

Create an object for performing meta-optimization.

Parameters

- **`learner`** (`MetaLearner`) – defines the meta-learning problem
- **`learning_rate`** (*`float or Tensor`*) – the learning rate to use for optimizing each task (not to be confused with the one used for meta-learning). This can optionally be made a variable (represented as a Tensor), in which case the learning rate will itself be learnable.
- **`optimization_steps`** (*`int`*) – the number of steps of gradient descent to perform for each task
- **`meta_batch_size`** (*`int`*) – the number of tasks to use for each step of meta-learning
- **`optimizer`** (`Optimizer`) – the optimizer to use for meta-learning (not to be confused with the gradient descent optimization performed for each task)
- **`model_dir`** (*`str`*) – the directory in which the model will be saved. If None, a temporary directory will be created.
- **`device`** (*`torch.device, optional (default None)`*) – the device on which to run computations. If None, a device is chosen automatically.

`fit`(*steps: int*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 600*, *restore: bool = False*)

Perform meta-learning to train the model.

Parameters

- **`steps`** (*`int`*) – the number of steps of meta-learning to perform
- **`max_checkpoints_to_keep`** (*`int`*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **`checkpoint_interval`** (*`int`*) – the time interval at which to save checkpoints, measured in seconds
- **`restore`** (*`bool`*) – if True, restore the model from the most recent checkpoint before training it further

restore() → None

Reload the model parameters from the most recent checkpoint file.

train_on_current_task(*optimization_steps: int = 1*, *restore: bool = True*)

Perform a few steps of gradient descent to fine tune the model on the current task.

Parameters

- **optimization_steps** (*int*) – the number of steps of gradient descent to perform
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint before optimizing

predict_on_batch(*inputs: Tensor | Sequence[Tensor]*) → Tuple[Tensor, Sequence[Tensor]]

Compute the model's outputs for a batch of inputs.

Parameters

inputs (*list of arrays*) – the inputs to the model

Returns

- *(loss, outputs) where loss is the value of the model's loss function, and*
- *outputs is a list of the model's outputs*

save_checkpoint(*max_checkpoints_to_keep: int = 5*, *model_dir: str | None = None*) → None

Save a checkpoint to disk.

Usually you do not need to call this method, since fit() saves checkpoints automatically. If you have disabled automatic checkpointing during fitting, this can be called to manually write checkpoints.

Parameters

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **model_dir** (*str, default None*) – Model directory to save checkpoint to. If None, revert to self.model_dir

get_checkpoints(*model_dir: str | None = None*) → List[str]

Get a list of all available checkpoint files.

Parameters

model_dir (*str, default None*) – Directory to get list of checkpoints from. Reverts to self.model_dir if None

3.29 Reinforcement Learning

Reinforcement Learning is a powerful technique for learning when you have access to a simulator. That is, suppose that you have a high fidelity way of predicting the outcome of an experiment. This is perhaps a physics engine, perhaps a chemistry engine, or anything. And you'd like to solve some task within this engine. You can use reinforcement learning for this purpose.

3.29.1 Environments

class Environment(*state_shape*, *n_actions=None*, *state_dtype=None*, *action_shape=None*)

An environment in which an actor performs actions to accomplish a task.

An environment has a current state, which is represented as either a single NumPy array, or optionally a list of NumPy arrays. When an action is taken, that causes the state to be updated. The environment also computes a reward for each action, and reports when the task has been terminated (meaning that no more actions may be taken).

Two types of actions are supported. For environments with discrete action spaces, the action is an integer specifying the index of the action to perform (out of a fixed list of possible actions). For environments with continuous action spaces, the action is a NumPy array.

Environment objects should be written to support pickle and deepcopy operations. Many algorithms involve creating multiple copies of the Environment, possibly running in different processes or even on different computers.

__init__(*state_shape*, *n_actions=None*, *state_dtype=None*, *action_shape=None*)

Subclasses should call the superclass constructor in addition to doing their own initialization.

A value should be provided for either n_actions (for discrete action spaces) or action_shape (for continuous action spaces), but not both.

Parameters

- **state_shape** (*tuple or list of tuples*) – the shape(s) of the array(s) making up the state
- **n_actions** (*int*) – the number of discrete actions that can be performed. If the action space is continuous, this should be None.
- **state_dtype** (*dtype or list of dtypes*) – the type(s) of the array(s) making up the state. If this is None, all arrays are assumed to be float32.
- **action_shape** (*tuple*) – the shape of the array describing an action. If the action space is discrete, this should be none.

property state

The current state of the environment, represented as either a NumPy array or list of arrays.

If reset() has not yet been called at least once, this is undefined.

property terminated

Whether the task has reached its end.

If reset() has not yet been called at least once, this is undefined.

property state_shape

The shape of the arrays that describe a state.

If the state is a single array, this returns a tuple giving the shape of that array. If the state is a list of arrays, this returns a list of tuples where each tuple is the shape of one array.

property state_dtype

The dtypes of the arrays that describe a state.

If the state is a single array, this returns the dtype of that array. If the state is a list of arrays, this returns a list containing the dtypes of the arrays.

`property n_actions`

The number of possible actions that can be performed in this Environment.

If the environment uses a continuous action space, this returns None.

`property action_shape`

The expected shape of NumPy arrays representing actions.

If the environment uses a discrete action space, this returns None.

`reset()`

Initialize the environment in preparation for doing calculations with it.

This must be called before calling step() or querying the state. You can call it again later to reset the environment back to its original state.

`step`(*action*)

Take a time step by performing an action.

This causes the "state" and "terminated" properties to be updated.

Parameters
`action` (*object*) – an object describing the action to take

Returns

- *the reward earned by taking the action, represented as a floating point number*
- *(higher values are better)*

`class GymEnvironment`(*name*)

This is a convenience class for working with environments from OpenAI Gym.

`__init__`(*name*)

Create an Environment wrapping the OpenAI Gym environment with a specified name.

`reset()`

Initialize the environment in preparation for doing calculations with it.

This must be called before calling step() or querying the state. You can call it again later to reset the environment back to its original state.

`step`(*action*)

Take a time step by performing an action.

This causes the "state" and "terminated" properties to be updated.

Parameters
`action` (*object*) – an object describing the action to take

Returns

- *the reward earned by taking the action, represented as a floating point number*
- *(higher values are better)*

3.29.2 Policies

class Policy(*output_names*, *rnn_initial_states=[]*)

A policy for taking actions within an environment.

A policy is defined by a tf.keras.Model that takes the current state as input and performs the necessary calculations. There are many algorithms for reinforcement learning, and they differ in what values they require a policy to compute. That makes it impossible to define a single interface allowing any policy to be optimized with any algorithm. Instead, this interface just tries to be as flexible and generic as possible. Each algorithm must document what values it expects the model to output.

Special handling is needed for models that include recurrent layers. In that case, the model has its own internal state which the learning algorithm must be able to specify and query. To support this, the Policy must do three things:

1. **The Model must take additional inputs that specify the initial states of**
   all its recurrent layers. These will be appended to the list of arrays specifying the environment state.
2. **The Model must also return the final states of all its recurrent layers as**
   outputs.
3. **The constructor argument rnn_initial_states must be specified to define**
   the states to use for the Model's recurrent layers at the start of a new rollout.

Policy objects should be written to support pickling. Many algorithms involve creating multiple copies of the Policy, possibly running in different processes or even on different computers.

__init__(*output_names*, *rnn_initial_states=[]*)

Subclasses should call the superclass constructor in addition to doing their own initialization.

Parameters

- **output_names** (*list of strings*) – the names of the Model's outputs, in order. It is up to each reinforcement learning algorithm to document what outputs it expects policies to compute. Outputs that return the final states of recurrent layers should have the name 'rnn_state'.
- **rnn_initial_states** (*list of NumPy arrays*) – the initial states of the Model's recurrent layers at the start of a new rollout

create_model(***kwargs*)

Construct and return a tf.keras.Model that computes the policy.

The inputs to the model consist of the arrays representing the current state of the environment, followed by the initial states for all recurrent layers. Depending on the algorithm being used, other inputs might get passed as well. It is up to each algorithm to document that.

3.29.3 Tensorflow implementation

3.29.4 A2C

class A2C(*env*, *policy*, *max_rollout_length=20*, *discount_factor=0.99*, *advantage_lambda=0.98*, *value_weight=1.0*, *entropy_weight=0.01*, *optimizer=None*, *model_dir=None*, *use_hindsight=False*)

Implements the Advantage Actor-Critic (A2C) algorithm for reinforcement learning.

The algorithm is described in Mnih et al, "Asynchronous Methods for Deep Reinforcement Learning" (https://arxiv.org/abs/1602.01783). This class supports environments with both discrete and continuous action spaces. For discrete action spaces, the "action" argument passed to the environment is an integer giving the index of the action to perform. The policy must output a vector called "action_prob" giving the probability of taking each action. For continuous action spaces, the action is an array where each element is chosen independently from a normal distribution. The policy must output two arrays of the same shape: "action_mean" gives the mean value for each element, and "action_std" gives the standard deviation for each element. In either case, the policy must also output a scalar called "value" which is an estimate of the value function for the current state.

The algorithm optimizes all outputs at once using a loss that is the sum of three terms:

1. The policy loss, which seeks to maximize the discounted reward for each action.
2. **The value loss, which tries to make the value estimate match the actual discounted reward**
   that was attained at each step.
3. An entropy term to encourage exploration.

This class supports Generalized Advantage Estimation as described in Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation" (https://arxiv.org/abs/1506.02438). This is a method of trading off bias and variance in the advantage estimate, which can sometimes improve the rate of convergance. Use the advantage_lambda parameter to adjust the tradeoff.

This class supports Hindsight Experience Replay as described in Andrychowicz et al., "Hindsight Experience Replay" (https://arxiv.org/abs/1707.01495). This is a method that can enormously accelerate learning when rewards are very rare. It requires that the environment state contains information about the goal the agent is trying to achieve. Each time it generates a rollout, it processes that rollout twice: once using the actual goal the agent was pursuing while generating it, and again using the final state of that rollout as the goal. This guarantees that half of all rollouts processed will be ones that achieved their goals, and hence received a reward.

To use this feature, specify use_hindsight=True to the constructor. The environment must have a method defined as follows:

def apply_hindsight(self, states, actions, goal):
   … return new_states, rewards

The method receives the list of states generated during the rollout, the action taken for each one, and a new goal state. It should generate a new list of states that are identical to the input ones, except specifying the new goal. It should return that list of states, and the rewards that would have been received for taking the specified actions from those states. The output arrays may be shorter than the input ones, if the modified rollout would have terminated sooner.

Note: Using this class on continuous action spaces requires that *tensorflow_probability* be installed.

`__init__`(*env*, *policy*, *max_rollout_length=20*, *discount_factor=0.99*, *advantage_lambda=0.98*, *value_weight=1.0*, *entropy_weight=0.01*, *optimizer=None*, *model_dir=None*, *use_hindsight=False*)

Create an object for optimizing a policy.

Parameters

- **`env`** (`Environment`) – the Environment to interact with
- **`policy`** (`Policy`) – the Policy to optimize. It must have outputs with the names 'action_prob' and 'value' (for discrete action spaces) or 'action_mean', 'action_std', and 'value' (for continuous action spaces)
- **`max_rollout_length`** (*`int`*) – the maximum length of rollouts to generate
- **`discount_factor`** (*`float`*) – the discount factor to use when computing rewards
- **`advantage_lambda`** (*`float`*) – the parameter for trading bias vs. variance in Generalized Advantage Estimation

- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **optimizer** (*Optimizer*) – the optimizer to use. If None, a default optimizer is used.
- **model_dir** (*str*) – the directory in which the model will be saved. If None, a temporary directory will be created.
- **use_hindsight** (*bool*) – if True, use Hindsight Experience Replay

fit(*total_steps*, *max_checkpoints_to_keep=5*, *checkpoint_interval=600*, *restore=False*)

Train the policy.

Parameters

- **total_steps** (*int*) – the total number of time steps to perform on the environment, across all rollouts on all threads
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **checkpoint_interval** (*float*) – the time interval at which to save checkpoints, measured in seconds
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

predict(*state*, *use_saved_states=True*, *save_states=True*)

Compute the policy's output predictions for a state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to generate predictions
- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the array of action probabilities, and the estimated value function

select_action(*state*, *deterministic=False*, *use_saved_states=True*, *save_states=True*)

Select an action to perform based on the environment's state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to select an action

- `deterministic` (*bool*) – if True, always return the best action (that is, the one with highest probability). If False, randomly select an action based on the computed probabilities.
- `use_saved_states` (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- `save_states` (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type
the index of the selected action

`restore()`

Reload the model parameters from the most recent checkpoint file.

`class A2CLossDiscrete`(*value_weight*, *entropy_weight*, *action_prob_index*, *value_index*)

This class computes the loss function for A2C with discrete action spaces.

`__init__`(*value_weight*, *entropy_weight*, *action_prob_index*, *value_index*)

3.29.5 PPO

`class PPO`(*env*, *policy*, *max_rollout_length=20*, *optimization_rollouts=8*, *optimization_epochs=4*, *batch_size=64*, *clipping_width=0.2*, *discount_factor=0.99*, *advantage_lambda=0.98*, *value_weight=1.0*, *entropy_weight=0.01*, *optimizer=None*, *model_dir=None*, *use_hindsight=False*)

Implements the Proximal Policy Optimization (PPO) algorithm for reinforcement learning.

The algorithm is described in Schulman et al, "Proximal Policy Optimization Algorithms" (https://openai-public.s3-us-west-2.amazonaws.com/blog/2017-07/ppo/ppo-arxiv.pdf). This class requires the policy to output two quantities: a vector giving the probability of taking each action, and an estimate of the value function for the current state. It optimizes both outputs at once using a loss that is the sum of three terms:

1. The policy loss, which seeks to maximize the discounted reward for each action.
2. **The value loss, which tries to make the value estimate match the actual discounted reward**
   that was attained at each step.
3. An entropy term to encourage exploration.

This class only supports environments with discrete action spaces, not continuous ones. The "action" argument passed to the environment is an integer, giving the index of the action to perform.

This class supports Generalized Advantage Estimation as described in Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation" (https://arxiv.org/abs/1506.02438). This is a method of trading off bias and variance in the advantage estimate, which can sometimes improve the rate of convergance. Use the advantage_lambda parameter to adjust the tradeoff.

This class supports Hindsight Experience Replay as described in Andrychowicz et al., "Hindsight Experience Replay" (https://arxiv.org/abs/1707.01495). This is a method that can enormously accelerate learning when rewards are very rare. It requires that the environment state contains information about the goal the agent is trying to achieve. Each time it generates a rollout, it processes that rollout twice: once using the actual goal the agent was pursuing while generating it, and again using the final state of that rollout as the goal. This guarantees that half of all rollouts processed will be ones that achieved their goals, and hence received a reward.

To use this feature, specify use_hindsight=True to the constructor. The environment must have a method defined as follows:

def apply_hindsight(self, states, actions, goal):
    … return new_states, rewards

The method receives the list of states generated during the rollout, the action taken for each one, and a new goal state. It should generate a new list of states that are identical to the input ones, except specifying the new goal. It should return that list of states, and the rewards that would have been received for taking the specified actions from those states. The output arrays may be shorter than the input ones, if the modified rollout would have terminated sooner.

`__init__`(*env*, *policy*, *max_rollout_length=20*, *optimization_rollouts=8*, *optimization_epochs=4*, *batch_size=64*, *clipping_width=0.2*, *discount_factor=0.99*, *advantage_lambda=0.98*, *value_weight=1.0*, *entropy_weight=0.01*, *optimizer=None*, *model_dir=None*, *use_hindsight=False*)

Create an object for optimizing a policy.

Parameters

- **env** (`Environment`) – the Environment to interact with
- **policy** (`Policy`) – the Policy to optimize. It must have outputs with the names 'action_prob' and 'value', corresponding to the action probabilities and value estimate
- **max_rollout_length** (*int*) – the maximum length of rollouts to generate
- **optimization_rollouts** (*int*) – the number of rollouts to generate for each iteration of optimization
- **optimization_epochs** (*int*) – the number of epochs of optimization to perform within each iteration
- **batch_size** (*int*) – the batch size to use during optimization. If this is 0, each rollout will be used as a separate batch.
- **clipping_width** (*float*) – in computing the PPO loss function, the probability ratio is clipped to the range (1-clipping_width, 1+clipping_width)
- **discount_factor** (*float*) – the discount factor to use when computing rewards
- **advantage_lambda** (*float*) – the parameter for trading bias vs. variance in Generalized Advantage Estimation
- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **optimizer** (`Optimizer`) – the optimizer to use. If None, a default optimizer is used.
- **model_dir** (*str*) – the directory in which the model will be saved. If None, a temporary directory will be created.
- **use_hindsight** (*bool*) – if True, use Hindsight Experience Replay

`fit`(*total_steps*, *max_checkpoints_to_keep=5*, *checkpoint_interval=600*, *restore=False*)

Train the policy.

Parameters

- **total_steps** (*int*) – the total number of time steps to perform on the environment, across all rollouts on all threads

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **checkpoint_interval** (*float*) – the time interval at which to save checkpoints, measured in seconds
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

predict(*state*, *use_saved_states=True*, *save_states=True*)

Compute the policy's output predictions for a state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to generate predictions
- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the array of action probabilities, and the estimated value function

select_action(*state*, *deterministic=False*, *use_saved_states=True*, *save_states=True*)

Select an action to perform based on the environment's state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to select an action
- **deterministic** (*bool*) – if True, always return the best action (that is, the one with highest probability). If False, randomly select an action based on the computed probabilities.
- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the index of the selected action

restore()

Reload the model parameters from the most recent checkpoint file.

class PPOLoss(*value_weight*, *entropy_weight*, *clipping_width*, *action_prob_index*, *value_index*)

This class computes the loss function for PPO.

__init__(*value_weight*, *entropy_weight*, *clipping_width*, *action_prob_index*, *value_index*)

3.29.6 Torch implementation

class A2CLossDiscrete(*value_weight: float*, *entropy_weight: float*, *action_prob_index: int*, *value_index: int*)

This class computes the loss function for A2C with discrete action spaces. The A2C algorithm optimizes all outputs at once using a loss that is the sum of three terms:

1. The policy loss, which seeks to maximize the discounted reward for each action.
2. The value loss, which tries to make the value estimate match the actual discounted reward that was attained at each step.
3. An entropy term to encourage exploration.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> outputs = [torch.tensor([[0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02,␣
↪0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02,0.02, 0.02, 0.02, 0.02, 0.02,
↪ 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.
↪2]]), torch.tensor([0.], requires_grad = True)]
>>> labels = np.array([[0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,␣
↪0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.
↪, 0., 0.]], dtype = np.float32)
>>> discount = np.array([-1.0203744, -0.02058018, 0.98931295, 2.009407, 1.019603, 0.
↪01980097, -0.9901, 0.01, -1.  , 0. ], dtype=np.float32)
>>> advantage = np.array([-1.0203744 ,-0.02058018, 0.98931295, 2.009407, 1.019603,␣
↪0.01980097, -0.9901 ,0.01 ,-1. , 0.], dtype = np.float32)
>>> loss = A2CLossDiscrete(value_weight = 1.0, entropy_weight = 0.01, action_prob_
↪index = 0, value_index = 1)
>>> loss_val = loss(outputs, [labels], [discount, advantage])
>>> loss_val
tensor(1.2541, grad_fn=<SubBackward0>)
```

__init__(*value_weight: float*, *entropy_weight: float*, *action_prob_index: int*, *value_index: int*)

Computes the loss function for the A2C algorithm with discrete action spaces.

Parameters

- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **action_prob_index** (*int*) – Index of the action probabilities in the model's outputs.
- **value_index** (*int*) – Index of the value estimate in the model's outputs.

class A2CLossContinuous(*value_weight: float*, *entropy_weight: float*, *mean_index: int*, *std_index: int*, *value_index: int*)

This class computes the loss function for A2C with continuous action spaces.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> outputs = [torch.tensor([[0.], [0.], [0.], [0.], [0.], [0.], [0.], [0.], [0.],␣
↪[0.]], dtype=torch.float32, requires_grad=True), torch.tensor([10.], dtype=torch.
↪float32, requires_grad=True), torch.tensor([[27.717865],[28.860144]], dtype=torch.
↪float32, requires_grad=True)]
>>> labels = np.array([[-4.897339 ], [ 3.4308329], [-4.527725 ], [-7.3000813], [-1.
↪9869075], [20.664988 ], [-8.448957 ], [10.580486 ], [10.017258 ], [17.884674 ]],␣
↪dtype=np.float32)
>>> discount = np.array([4.897339, -8.328172, 7.958559, 2.772356, -5.313174, -22.
↪651896, 29.113945, -19.029444, 0.56322646, -7.867417], dtype=np.float32)
>>> advantage = np.array([-5.681633, -20.57494, -1.4520378, -9.348538, -18.378199, -
↪33.907513, 25.572464, -32.485718 , -6.412546, -15.034998], dtype=np.float32)
>>> loss = A2CLossContinuous(value_weight = 1.0, entropy_weight = 0.01, mean_index␣
↪= 0, std_index = 1, value_index = 2)
>>> loss_val = loss(outputs, [labels], [discount, advantage])
>>> loss_val
tensor(1050.2310, grad_fn=<SubBackward0>)
```

__init__(*value_weight: float*, *entropy_weight: float*, *mean_index: int*, *std_index: int*, *value_index: int*)

Computes the loss function for the A2C algorithm with continuous action spaces.

Parameters

- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **mean_index** (*int*) – Index of the mean of the action distribution in the model's outputs.
- **std_index** (*int*) – Index of the standard deviation of the action distribution in the model's outputs.
- **value_index** (*int*) – Index of the value estimate in the model's outputs.

3.29.7 A2C

class A2C(*env: Environment*, *policy: Policy*, *max_rollout_length: int = 20*, *discount_factor: float = 0.99*, *advantage_lambda: float = 0.98*, *value_weight: float = 1.0*, *entropy_weight: float = 0.01*, *optimizer: Optimizer | None = None*, *model_dir: str | None = None*, *use_hindsight: bool = False*, *device: device | None = None*)

Implements the Advantage Actor-Critic (A2C) algorithm for reinforcement learning. The algorithm is described in Mnih et al, "Asynchronous Methods for Deep Reinforcement Learning" (https://arxiv.org/abs/1602.01783). This class supports environments with both discrete and continuous action spaces. For discrete action spaces, the "action" argument passed to the environment is an integer giving the index of the action to perform. The policy must output a vector called "action_prob" giving the probability of taking each action. For continuous action spaces, the action is an array where each element is chosen independently from a normal distribution. The policy must output two arrays of the same shape: "action_mean" gives the mean value for each element, and "action_std" gives the standard deviation for each element. In either case, the policy must also output a scalar called "value" which is an estimate of the value function for the current state. The algorithm optimizes all outputs at once using a loss that is the sum of three terms:

1. The policy loss, which seeks to maximize the discounted reward for each action.
2. **The value loss, which tries to make the value estimate match the actual discounted reward**
   that was attained at each step.
3. An entropy term to encourage exploration.

This class supports Generalized Advantage Estimation as described in Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation" (https://arxiv.org/abs/1506.02438). This is a method of trading off bias and variance in the advantage estimate, which can sometimes improve the rate of convergance. Use the advantage_lambda parameter to adjust the tradeoff. This class supports Hindsight Experience Replay as described in Andrychowicz et al., "Hindsight Experience Replay" (https://arxiv.org/abs/1707.01495). This is a method that can enormously accelerate learning when rewards are very rare. It requires that the environment state contains information about the goal the agent is trying to achieve. Each time it generates a rollout, it processes that rollout twice: once using the actual goal the agent was pursuing while generating it, and again using the final state of that rollout as the goal. This guarantees that half of all rollouts processed will be ones that achieved their goals, and hence received a reward. To use this feature, specify use_hindsight=True to the constructor. The environment must have a method defined as follows:

def apply_hindsight(self, states, actions, goal):
    … return new_states, rewards

The method receives the list of states generated during the rollout, the action taken for each one, and a new goal state. It should generate a new list of states that are identical to the input ones, except specifying the new goal. It should return that list of states, and the rewards that would have been received for taking the specified actions from those states. The output arrays may be shorter than the input ones, if the modified rollout would have terminated sooner.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> from deepchem.rl.torch_rl import A2C
>>> class RouletteEnvironment(dc.rl.Environment):
...     def __init__(self):
...         super(RouletteEnvironment, self).__init__([(1,)], 38)
...         self._state = [np.array([0])]
...     def step(self, action):
...         if action == 37:
...             self._terminated = True  # Walk away.
...             return 0.0
...         wheel = np.random.randint(37)
...         if wheel == 0:
...             if action == 0:
...                 return 35.0
...             return -1.0
...         if action != 0 and wheel % 2 == action % 2:
```

(continues on next page)

(continued from previous page)

```
...             return 1.0
...         return -1.0
...     def reset(self):
...         self._terminated = False
>>> class TestPolicy(dc.rl.Policy):
...     def __init__(self, env):
...         super(TestPolicy, self).__init__(['action_prob', 'value'])
...         self.env = env
...     def create_model(self, **kwargs):
...         env = self.env
...         class TestModel(nn.Module):
...             def __init__(self):
...                 super(TestModel, self).__init__()
...                 self.action = nn.Parameter(torch.ones(env.n_actions,␣
↪dtype=torch.float32))
...                 self.value = nn.Parameter(torch.tensor([0.0], dtype=torch.
↪float32))
...             def forward(self, inputs):
...                 prob = F.softmax(torch.reshape(self.action, (-1, env.n_
↪actions)))
...                 return prob, self.value
...         return TestModel()
>>> env = RouletteEnvironment()
>>> policy = TestPolicy(env)
>>> a2c = A2C(env, policy, max_rollout_length=20, optimizer=Adam(learning_rate=0.
↪001))
>>> a2c.fit(1000)
>>> action_prob, value = a2c.predict([[0]])
```

`__init__`(*env: Environment*, *policy: Policy*, *max_rollout_length: int = 20*, *discount_factor: float = 0.99*, *advantage_lambda: float = 0.98*, *value_weight: float = 1.0*, *entropy_weight: float = 0.01*, *optimizer: Optimizer | None = None*, *model_dir: str | None = None*, *use_hindsight: bool = False*, *device: device | None = None*) → None

Create an object for optimizing a policy.

Parameters

- **env** (`Environment`) – the Environment to interact with
- **policy** (`Policy`) – the Policy to optimize. It must have outputs with the names 'action_prob' and 'value' (for discrete action spaces) or 'action_mean', 'action_std', and 'value' (for continuous action spaces)
- **max_rollout_length** (*int*) – the maximum length of rollouts to generate
- **discount_factor** (*float*) – the discount factor to use when computing rewards
- **advantage_lambda** (*float*) – the parameter for trading bias vs. variance in Generalized Advantage Estimation
- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **optimizer** (`Optimizer`) – the optimizer to use. If None, a default optimizer is used.

- **`model_dir`** (*`str`*) – the directory in which the model will be saved. If None, a temporary directory will be created.
- **`use_hindsight`** (*`bool`*) – if True, use Hindsight Experience Replay
- **`device`** (*`torch.device, optional (default None)`*) – the device on which to run computations. If None, a device is chosen automatically.

`fit`(*total_steps: int*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 600*, *restore: bool = False*) → None

Train the policy.

Parameters

- **`total_steps`** (*`int`*) – the total number of time steps to perform on the environment, across all rollouts on all threads
- **`max_checkpoints_to_keep`** (*`int`*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **`checkpoint_interval`** (*`float`*) – the time interval at which to save checkpoints, measured in seconds
- **`restore`** (*`bool`*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

`predict`(*state: ndarray*, *use_saved_states: bool = True*, *save_states: bool = True*) → List[ndarray]

Compute the policy's output predictions for a state. If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **`state`** (*`array or list of arrays`*) – the state of the environment for which to generate predictions
- **`use_saved_states`** (*`bool`*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **`save_states`** (*`bool`*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the array of action probabilities, and the estimated value function

`select_action`(*state: List[ndarray]*, *deterministic: bool = False*, *use_saved_states: bool = True*, *save_states: bool = True*) → int

Select an action to perform based on the environment's state. If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **`state`** (*`array or list of arrays`*) – the state of the environment for which to select an action
- **`deterministic`** (*`bool`*) – if True, always return the best action (that is, the one with highest probability). If False, randomly select an action based on the computed probabilities.

- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the index of the selected action

restore(*strict: bool | None = True*) → None

Reload the model parameters from the most recent checkpoint file.

save_checkpoint(*max_checkpoints_to_keep: int = 5*, *model_dir: str | None = None*) → None

Save a checkpoint to disk. Usually you do not need to call this method, since fit() saves checkpoints automatically. If you have disabled automatic checkpointing during fitting, this can be called to manually write checkpoints.

Parameters

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **model_dir** (*str, default None*) – Model directory to save checkpoint to. If None, revert to self.model_dir

get_checkpoints(*model_dir: str | None = None*) → List[str]

Get a list of all available checkpoint files.

Parameters

model_dir (*str, default None*) – Directory to get list of checkpoints from. Reverts to self.model_dir if None

3.29.8 PPO

class PPO(*env: Environment*, *policy: Policy*, *max_rollout_length: int = 20*, *optimization_rollouts: int = 8*, *optimization_epochs: int = 4*, *batch_size: int = 64*, *clipping_width: float = 0.2*, *discount_factor: float = 0.99*, *advantage_lambda: float = 0.98*, *value_weight: float = 1.0*, *entropy_weight: float = 0.01*, *optimizer: Optimizer | None = None*, *model_dir: str | None = None*, *use_hindsight: bool = False*, *device: device | None = None*)

Implements the Proximal Policy Optimization (PPO) algorithm for reinforcement learning.

The algorithm is described in Schulman et al, “Proximal Policy Optimization Algorithms” (https://openai-public.s3-us-west-2.amazonaws.com/blog/2017-07/ppo/ppo-arxiv.pdf). This class requires the policy to output two quantities: a vector giving the probability of taking each action, and an estimate of the value function for the current state. It optimizes both outputs at once using a loss that is the sum of three terms:

1. The policy loss, which seeks to maximize the discounted reward for each action.
2. **The value loss, which tries to make the value estimate match the actual discounted reward**
   that was attained at each step.
3. An entropy term to encourage exploration.

This class only supports environments with discrete action spaces, not continuous ones. The “action” argument passed to the environment is an integer, giving the index of the action to perform.

This class supports Generalized Advantage Estimation as described in Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation" (https://arxiv.org/abs/1506.02438). This is a method of trading off bias and variance in the advantage estimate, which can sometimes improve the rate of convergance. Use the advantage_lambda parameter to adjust the tradeoff.

This class supports Hindsight Experience Replay as described in Andrychowicz et al., "Hindsight Experience Replay" (https://arxiv.org/abs/1707.01495). This is a method that can enormously accelerate learning when rewards are very rare. It requires that the environment state contains information about the goal the agent is trying to achieve. Each time it generates a rollout, it processes that rollout twice: once using the actual goal the agent was pursuing while generating it, and again using the final state of that rollout as the goal. This guarantees that half of all rollouts processed will be ones that achieved their goals, and hence received a reward.

To use this feature, specify use_hindsight=True to the constructor. The environment must have a method defined as follows:

def apply_hindsight(self, states, actions, goal):
    … return new_states, rewards

The method receives the list of states generated during the rollout, the action taken for each one, and a new goal state. It should generate a new list of states that are identical to the input ones, except specifying the new goal. It should return that list of states, and the rewards that would have been received for taking the specified actions from those states. The output arrays may be shorter than the input ones, if the modified rollout would have terminated sooner.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> from deepchem.rl.torch_rl import PPO
>>> class RouletteEnvironment(dc.rl.Environment):
...     def __init__(self):
...         super(RouletteEnvironment, self).__init__([(1,)], 38)
...         self._state = [np.array([0])]
...     def step(self, action):
...         if action == 37:
...             self._terminated = True  # Walk away.
...             return 0.0
...         wheel = np.random.randint(37)
...         if wheel == 0:
...             if action == 0:
...                 return 35.0
...             return -1.0
...         if action != 0 and wheel % 2 == action % 2:
...             return 1.0
...         return -1.0
...     def reset(self):
...         self._terminated = False
>>> class TestPolicy(dc.rl.Policy):
...     def __init__(self):
...         super(TestPolicy, self).__init__(['action_prob', 'value'])
...     def create_model(self, **kwargs):
...         class TestModel(nn.Module):
```

(continues on next page)

(continued from previous page)

```
...             def __init__(self):
...                 super(TestModel, self).__init__()
...                 self.action = nn.Parameter(torch.ones(env.n_actions,␣
↪dtype=torch.float32))
...                 self.value = nn.Parameter(torch.tensor([0.0], dtype=torch.
↪float32))
...             def forward(self, inputs):
...                 prob = F.softmax(torch.reshape(self.action, (-1, env.n_
↪actions)))
...                 return prob, self.value
...         return TestModel()
>>> env = RouletteEnvironment()
>>> ppo = PPO(env, TestPolicy(), max_rollout_length=20, optimization_epochs=8,␣
↪optimizer=Adam(learning_rate=0.003))
>>> ppo.fit(100000)
>>> action_prob, value = ppo.predict([[0]])
```

__init__(*env: Environment*, *policy: Policy*, *max_rollout_length: int = 20*, *optimization_rollouts: int = 8*, *optimization_epochs: int = 4*, *batch_size: int = 64*, *clipping_width: float = 0.2*, *discount_factor: float = 0.99*, *advantage_lambda: float = 0.98*, *value_weight: float = 1.0*, *entropy_weight: float = 0.01*, *optimizer: Optimizer | None = None*, *model_dir: str | None = None*, *use_hindsight: bool = False*, *device: device | None = None*) → None

Create an object for optimizing a policy.

Parameters

- **env** (Environment) – the Environment to interact with
- **policy** (Policy) – the Policy to optimize. It must have outputs with the names 'action_prob' and 'value', corresponding to the action probabilities and value estimate
- **max_rollout_length** (*int*) – the maximum length of rollouts to generate
- **optimization_rollouts** (*int*) – the number of rollouts to generate for each iteration of optimization
- **optimization_epochs** (*int*) – the number of epochs of optimization to perform within each iteration
- **batch_size** (*int*) – the batch size to use during optimization. If this is 0, each rollout will be used as a separate batch.
- **clipping_width** (*float*) – in computing the PPO loss function, the probability ratio is clipped to the range (1-clipping_width, 1+clipping_width)
- **discount_factor** (*float*) – the discount factor to use when computing rewards
- **advantage_lambda** (*float*) – the parameter for trading bias vs. variance in Generalized Advantage Estimation
- **value_weight** (*float*) – a scale factor for the value loss term in the loss function
- **entropy_weight** (*float*) – a scale factor for the entropy term in the loss function
- **optimizer** (Optimizer) – the optimizer to use. If None, a default optimizer is used.
- **model_dir** (*str*) – the directory in which the model will be saved. If None, a temporary directory will be created.
- **use_hindsight** (*bool*) – if True, use Hindsight Experience Replay

`fit`(*total_steps: int*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 600*, *restore: bool = False*) → None

Train the policy.

Parameters

- **total_steps** (*int*) – the total number of time steps to perform on the environment, across all rollouts on all threads
- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoint files to keep. When this number is reached, older files are deleted.
- **checkpoint_interval** (*float*) – the time interval at which to save checkpoints, measured in seconds
- **restore** (*bool*) – if True, restore the model from the most recent checkpoint and continue training from there. If False, retrain the model from scratch.

`predict`(*state: ndarray*, *use_saved_states: bool = True*, *save_states: bool = True*) → List[ndarray]

Compute the policy's output predictions for a state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to generate predictions
- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.
- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the array of action probabilities, and the estimated value function

`select_action`(*state: List[ndarray]*, *deterministic: bool = False*, *use_saved_states: bool = True*, *save_states: bool = True*) → int

Select an action to perform based on the environment's state.

If the policy involves recurrent layers, this method can preserve their internal states between calls. Use the use_saved_states and save_states arguments to specify how it should behave.

Parameters

- **state** (*array or list of arrays*) – the state of the environment for which to select an action
- **deterministic** (*bool*) – if True, always return the best action (that is, the one with highest probability). If False, randomly select an action based on the computed probabilities.
- **use_saved_states** (*bool*) – if True, the states most recently saved by a previous call to predict() or select_action() will be used as the initial states. If False, the internal states of all recurrent layers will be set to the initial values defined by the policy before computing the predictions.

- **save_states** (*bool*) – if True, the internal states of all recurrent layers at the end of the calculation will be saved, and any previously saved states will be discarded. If False, the states at the end of the calculation will be discarded, and any previously saved states will be kept.

Return type

the index of the selected action

restore(*strict: bool | None = True*) → None

Reload the model parameters from the most recent checkpoint file.

save_checkpoint(*max_checkpoints_to_keep: int = 5*, *model_dir: str | None = None*) → None

Save a checkpoint to disk.

Usually you do not need to call this method, since fit() saves checkpoints automatically. If you have disabled automatic checkpointing during fitting, this can be called to manually write checkpoints.

Parameters

- **max_checkpoints_to_keep** (*int*) – the maximum number of checkpoints to keep. Older checkpoints are discarded.
- **model_dir** (*str, default None*) – Model directory to save checkpoint to. If None, revert to self.model_dir

get_checkpoints(*model_dir: str | None = None*) → List[str]

Get a list of all available checkpoint files.

Parameters

model_dir (*str, default None*) – Directory to get list of checkpoints from. Reverts to self.model_dir if None

class PPOLoss(*value_weight: float*, *entropy_weight: float*, *clipping_width: float*, *action_prob_index: int*, *value_index: int*)

This class computes the loss function for PPO.

Example

```
>>> import deepchem as dc
>>> import numpy as np
>>> import torch
>>> import torch.nn.functional as F
>>> outputs = [torch.tensor([[0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02,␣
↪0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.02,0.02, 0.02, 0.02, 0.02, 0.02,
↪ 0.02, 0.02, 0.02, 0.02, 0.02, 0.02, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.
↪2]]), torch.tensor([0.], requires_grad = True)]
>>> labels = np.array([[0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,␣
↪0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.
↪, 0., 0.]], dtype = np.float32)
>>> discount = np.array([-1.0203744, -0.02058018, 0.98931295, 2.009407, 1.019603, 0.
↪01980097, -0.9901, 0.01, -1. , 0. ], dtype=np.float32)
>>> advantage = np.array([-1.0203744 ,-0.02058018, 0.98931295, 2.009407, 1.019603,␣
↪0.01980097, -0.9901 ,0.01 ,-1. , 0.], dtype = np.float32)
>>> old_prob = np.array([0.28183755, 0.95147914, 0.87922776, 0.8037652 , 0.11757819,
↪ 0.271103  , 0.21057394, 0.78721744, 0.6545527 , 0.8832647 ], dtype=np.float32)
>>> loss = PPOLoss(value_weight = 1.0, entropy_weight = 0.01, clipping_width = 0.2,␣
```

(continues on next page)

(continued from previous page)

```
↪action_prob_index = 0, value_index = 1)
>>> loss_val = loss(outputs, [labels], [discount, advantage, old_prob])
>>> loss_val
tensor(1.0761, grad_fn=<SubBackward0>)
```

`__init__`(*value_weight: float*, *entropy_weight: float*, *clipping_width: float*, *action_prob_index: int*, *value_index: int*)

3.30 Docking

Thanks to advances in biophysics, we are often able to find the structure of proteins from experimental techniques like Cryo-EM or X-ray crystallography. These structures can be powerful aides in designing small molecules. The technique of Molecular docking performs geometric calculations to find a "binding pose" with the small molecule interacting with the protein in question in a suitable binding pocket (that is, a region on the protein which has a groove in which the small molecule can rest). For more information about docking, check out the Autodock Vina paper:

Trott, Oleg, and Arthur J. Olson. "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading." Journal of computational chemistry 31.2 (2010): 455-461.

3.30.1 Binding Pocket Discovery

DeepChem has some utilities to help find binding pockets on proteins automatically. For now, these utilities are simple, but we will improve these in future versions of DeepChem.

`class` **`BindingPocketFinder`**

Abstract superclass for binding pocket detectors

Many times when working with a new protein or other macromolecule, it's not clear what zones of the macromolecule may be good targets for potential ligands or other molecules to interact with. This abstract class provides a template for child classes that algorithmically locate potential binding pockets that are good potential interaction sites.

Note that potential interactions sites can be found by many different methods, and that this abstract class doesn't specify the technique to be used.

`find_pockets`(*molecule: Any*)

Finds potential binding pockets in proteins.

Parameters

`molecule` (*`object`*) – Some representation of a molecule.

`class` **`ConvexHullPocketFinder`**(*scoring_model: Model | None = None*, *pad: float = 5.0*)

Implementation that uses convex hull of protein to find pockets.

Based on https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4112621/pdf/1472-6807-14-18.pdf

`__init__`(*scoring_model: Model | None = None*, *pad: float = 5.0*)

Initialize the pocket finder.

Parameters

- **`scoring_model`** (`Model`, *`optional (default None)`*) – If specified, use this model to prune pockets.

- **pad** (*float, optional (default 5.0)*) – The number of angstroms to pad around a binding pocket's atoms to get a binding pocket box.

find_all_pockets(*protein_file: str*) → List[*CoordinateBox*]

Find list of binding pockets on protein.

Parameters
protein_file (*str*) – Protein to load in.

Returns
List of binding pockets on protein. Each pocket is a *CoordinateBox*

Return type
List[*CoordinateBox*]

find_pockets(*macromolecule_file: str*) → List[*CoordinateBox*]

Find list of suitable binding pockets on protein.

This function computes putative binding pockets on this protein. This class uses the *ConvexHull* to compute binding pockets. Each face of the hull is converted into a coordinate box used for binding.

Parameters
macromolecule_file (*str*) – Location of the macromolecule file to load

Returns
List of pockets. Each pocket is a *CoordinateBox*

Return type
List[*CoordinateBox*]

3.30.2 Pose Generation

Pose generation is the task of finding a "pose", that is a geometric configuration of a small molecule interacting with a protein. Pose generation is a complex process, so for now DeepChem relies on external software to perform pose generation. This software is invoked and installed under the hood.

class PoseGenerator

A Pose Generator computes low energy conformations for molecular complexes.

Many questions in structural biophysics reduce to that of computing the binding free energy of molecular complexes. A key step towards computing the binding free energy of two complexes is to find low energy "poses", that is energetically favorable conformations of molecules with respect to each other. One application of this technique is to find low energy poses for protein-ligand interactions.

generate_poses(*molecular_complex: Tuple[str, str]*, *centroid: ndarray | None = None*, *box_dims: ndarray | None = None*, *exhaustiveness: int = 10*, *num_modes: int = 9*, *num_pockets: int | None = None*, *out_dir: str | None = None*, *generate_scores: bool = False*)

Generates a list of low energy poses for molecular complex

Parameters

- **molecular_complexes** (*Tuple[str, str]*) – A representation of a molecular complex. This tuple is (protein_file, ligand_file).
- **centroid** (*np.ndarray, optional (default None)*) – The centroid to dock against. Is computed if not specified.
- **box_dims** (*np.ndarray, optional (default None)*) – A numpy array of shape *(3,)* holding the size of the box to dock. If not specified is set to size of molecular complex plus 5 angstroms.

- **exhaustiveness** (*int, optional (default 10)*) – Tells pose generator how exhaustive it should be with pose generation.
- **num_modes** (*int, optional (default 9)*) – Tells pose generator how many binding modes it should generate at each invocation.
- **num_pockets** (*int, optional (default None)*) – If specified, *self.pocket_finder* must be set. Will only generate poses for the first *num_pockets* returned by *self.pocket_finder*.
- **out_dir** (*str, optional (default None)*) – If specified, write generated poses to this directory.
- **generate_score** (*bool, optional (default False)*) – If *True*, the pose generator will return scores for complexes. This is used typically when invoking external docking programs that compute scores.

Return type

A list of molecular complexes in energetically favorable poses.

`class VinaPoseGenerator`(*pocket_finder: BindingPocketFinder | None = None*)

Uses Autodock Vina to generate binding poses.

This class uses Autodock Vina to make make predictions of binding poses.

Example

>> import deepchem as dc >> vpg = dc.dock.VinaPoseGenerator(pocket_finder=None) >> protein_file = '1jld_protein.pdb' >> ligand_file = '1jld_ligand.sdf' >> poses, scores = vpg.generate_poses(.. (protein_file, ligand_file), .. exhaustiveness=1, .. num_modes=1, .. out_dir=tmp, .. generate_scores=True)

Note: This class requires RDKit and vina to be installed. As on 9-March-22, Vina is not available on Windows. Hence, this utility is currently available only on Ubuntu and MacOS.

`__init__`(*pocket_finder: BindingPocketFinder | None = None*)

Initializes Vina Pose Generator

Parameters

pocket_finder (*BindingPocketFinder, optional (default None)*) – If specified should be an instance of *dc.dock.BindingPocketFinder*.

`generate_poses`(*molecular_complex: Tuple[str, str], centroid: ndarray | None = None, box_dims: ndarray | None = None, exhaustiveness: int = 10, num_modes: int = 9, num_pockets: int | None = None, out_dir: str | None = None, generate_scores: bool | None = False, **kwargs*) → Tuple[List[Tuple[Any, Any]], List[float]] | List[Tuple[Any, Any]]

Generates the docked complex and outputs files for docked complex.

Parameters

- **molecular_complexes** (*Tuple[str, str]*) – A representation of a molecular complex. This tuple is (protein_file, ligand_file). The protein should be a pdb file and the ligand should be an sdf file.
- **centroid** (*np.ndarray, optional*) – The centroid to dock against. Is computed if not specified.
- **box_dims** (*np.ndarray, optional*) – A numpy array of shape *(3,)* holding the size of the box to dock. If not specified is set to size of molecular complex plus 5 angstroms.

- **exhaustiveness** (*int, optional (default 10)*) – Tells Autodock Vina how exhaustive it should be with pose generation. A higher value of exhaustiveness implies more computation effort for the docking experiment.
- **num_modes** (*int, optional (default 9)*) – Tells Autodock Vina how many binding modes it should generate at each invocation.
- **num_pockets** (*int, optional (default None)*) – If specified, *self.pocket_finder* must be set. Will only generate poses for the first *num_pockets* returned by *self.pocket_finder*.
- **out_dir** (*str, optional*) – If specified, write generated poses to this directory.
- **generate_score** (*bool, optional (default False)*) – If *True*, the pose generator will return scores for complexes. This is used typically when invoking external docking programs that compute scores.
- **kwargs** – The kwargs - cpu, min_rmsd, max_evals, energy_range supported by VINA are as documented in https://autodock-vina.readthedocs.io/en/latest/vina.html

Returns
Tuple of *(docked_poses, scores)*, *docked_poses*, or *scores*. *docked_poses* is a list of docked molecular complexes. Each entry in this list contains a *(protein_mol, ligand_mol)* pair of RDKit molecules. *scores* is a list of binding free energies predicted by Vina.

Return type
Tuple[*docked_poses*, *scores*] or *docked_poses* or *scores*

Raises
`ValueError` –

`class GninaPoseGenerator`

Use GNINA to generate binding poses.

This class uses GNINA (a deep learning framework for molecular docking) to generate binding poses. It downloads the GNINA executable to DEEPCHEM_DATA_DIR (an environment variable you set) and invokes the executable to perform pose generation.

GNINA uses pre-trained convolutional neural network (CNN) scoring functions to rank binding poses based on learned representations of 3D protein-ligand interactions. It has been shown to outperform AutoDock Vina in virtual screening applications [1]_.

If you use the GNINA molecular docking engine, please cite the relevant papers: https://github.com/gnina/gnina#citation The primary citation for GNINA is [1]_.

References

"Protein–Ligand Scoring with Convolutional Neural Networks." Journal of chemical information and modeling (2017).

Note:

- GNINA currently only works on Linux operating systems.
- GNINA requires CUDA >= 10.1 for fast CNN scoring.
- **Almost all dependencies are included in the most compatible way**
  possible, which reduces performance. Build GNINA from source for production use.

`__init__()`

Initialize GNINA pose generator.

`generate_poses`(*molecular_complex: Tuple[str, str]*, *centroid: ndarray | None = None*, *box_dims: ndarray | None = None*, *exhaustiveness: int = 10*, *num_modes: int = 9*, *num_pockets: int | None = None*, *out_dir: str | None = None*, *generate_scores: bool = True*, ***kwargs*) → Tuple[List[Tuple[Any, Any]], ndarray] | List[Tuple[Any, Any]]

Generates the docked complex and outputs files for docked complex.

Parameters

- **molecular_complexes** (*Tuple[str, str]*) – A representation of a molecular complex. This tuple is (protein_file, ligand_file).
- **centroid** (*np.ndarray, optional (default None)*) – The centroid to dock against. Is computed if not specified.
- **box_dims** (*np.ndarray, optional (default None)*) – A numpy array of shape *(3,)* holding the size of the box to dock. If not specified is set to size of molecular complex plus 4 angstroms.
- **exhaustiveness** (*int (default 8)*) – Tells GNINA how exhaustive it should be with pose generation.
- **num_modes** (*int (default 9)*) – Tells GNINA how many binding modes it should generate at each invocation.
- **out_dir** (*str, optional*) – If specified, write generated poses to this directory.
- **generate_scores** (*bool, optional (default True)*) – If *True*, the pose generator will return scores for complexes. This is used typically when invoking external docking programs that compute scores.
- **kwargs** – Any args supported by GNINA as documented https://github.com/gnina/gnina#usage

Returns

Tuple of *(docked_poses, scores)* or *docked_poses*. *docked_poses* is a list of docked molecular complexes. Each entry in this list contains a *(protein_mol, ligand_mol)* pair of RDKit molecules. *scores* is an array of binding affinities (kcal/mol), CNN pose scores, and CNN affinities predicted by GNINA.

Return type

Tuple[*docked_poses*, *scores*] or *docked_poses*

3.30.3 Docking

The `dc.dock.docking` module provides a generic docking implementation that depends on provide pose generation and pose scoring utilities to perform docking. This implementation is generic.

`class Docker`(*pose_generator:* PoseGenerator, *featurizer:* ComplexFeaturizer | *None = None*, *scoring_model:* Model | *None = None*)

A generic molecular docking class

This class provides a docking engine which uses provided models for featurization, pose generation, and scoring. Most pieces of docking software are command line tools that are invoked from the shell. The goal of this class is to provide a python clean API for invoking molecular docking programmatically.

The implementation of this class is lightweight and generic. It's expected that the majority of the heavy lifting will be done by pose generation and scoring classes that are provided to this class.

`__init__`(*pose_generator:* PoseGenerator, *featurizer:* ComplexFeaturizer | *None* = *None*, *scoring_model:* Model | *None* = *None*)

Builds model.

Parameters

- **pose_generator** (`PoseGenerator`) – The pose generator to use for this model
- **featurizer** (`ComplexFeaturizer`, *optional (default None)*) – Featurizer associated with *scoring_model*
- **scoring_model** (`Model`, *optional (default None)*) – Should make predictions on molecular complex.

`dock`(*molecular_complex: Tuple[str, str]*, *centroid: ndarray* | *None* = *None*, *box_dims: ndarray* | *None* = *None*, *exhaustiveness: int* = *10*, *num_modes: int* = *9*, *num_pockets: int* | *None* = *None*, *out_dir: str* | *None* = *None*, *use_pose_generator_scores: bool* = *False*) → Generator[Tuple[Any, Any], None, None] | Generator[Tuple[Tuple[Any, Any], float], None, None]

Generic docking function.

This docking function uses this object's featurizer, pose generator, and scoring model to make docking predictions. This function is written in generic style so

Parameters

- **molecular_complex** (*Tuple[str, str]*) – A representation of a molecular complex. This tuple is (protein_file, ligand_file).
- **centroid** (*np.ndarray, optional (default None)*) – The centroid to dock against. Is computed if not specified.
- **box_dims** (*np.ndarray, optional (default None)*) – A numpy array of shape *(3,)* holding the size of the box to dock. If not specified is set to size of molecular complex plus 5 angstroms.
- **exhaustiveness** (*int, optional (default 10)*) – Tells pose generator how exhaustive it should be with pose generation.
- **num_modes** (*int, optional (default 9)*) – Tells pose generator how many binding modes it should generate at each invocation.
- **num_pockets** (*int, optional (default None)*) – If specified, *self.pocket_finder* must be set. Will only generate poses for the first *num_pockets* returned by *self.pocket_finder*.
- **out_dir** (*str, optional (default None)*) – If specified, write generated poses to this directory.
- **use_pose_generator_scores** (*bool, optional (default False)*) – If *True*, ask pose generator to generate scores. This cannot be *True* if *self.featurizer* and *self.scoring_model* are set since those will be used to generate scores in that case.

Returns

A generator. If *use_pose_generator_scores==True* or *self.scoring_model* is set, then will yield tuples *(posed_complex, score)*. Else will yield *posed_complex*.

Return type

Generator[Tuple[*posed_complex*, *score*]] or Generator[*posed_complex*]

3.30.4 Pose Scoring

This module contains some utilities for computing docking scoring functions directly in Python. For now, support for custom pose scoring is limited.

pairwise_distances(*coords1: ndarray*, *coords2: ndarray*) → ndarray

Returns matrix of pairwise Euclidean distances.

Parameters

- **coords1** (*np.ndarray*) – A numpy array of shape *(N, 3)*
- **coords2** (*np.ndarray*) – A numpy array of shape *(M, 3)*

Returns
A *(N,M)* array with pairwise distances.

Return type
np.ndarray

cutoff_filter(*d: ndarray*, *x: ndarray*, *cutoff=8.0*) → ndarray

Applies a cutoff filter on pairwise distances

Parameters

- **d** (*np.ndarray*) – Pairwise distances matrix. A numpy array of shape *(N, M)*
- **x** (*np.ndarray*) – Matrix of shape *(N, M)*
- **cutoff** (*float, optional (default 8)*) – Cutoff for selection in Angstroms

Returns
A *(N,M)* array with values where distance is too large thresholded to 0.

Return type
np.ndarray

vina_nonlinearity(*c: ndarray*, *w: float*, *Nrot: int*) → ndarray

Computes non-linearity used in Vina.

Parameters

- **c** (*np.ndarray*) – A numpy array of shape *(N, M)*
- **w** (*float*) – Weighting term
- **Nrot** (*int*) – Number of rotatable bonds in this molecule

Returns
A *(N, M)* array with activations under a nonlinearity.

Return type
np.ndarray

vina_repulsion(*d: ndarray*) → ndarray

Computes Autodock Vina's repulsion interaction term.

Parameters
d (*np.ndarray*) – A numpy array of shape *(N, M)*.

Returns
A *(N, M)* array with repulsion terms.

Return type
np.ndarray

vina_hydrophobic(*d: ndarray*) → ndarray

Computes Autodock Vina's hydrophobic interaction term.

Here, d is the set of surface distances as defined in [1]_

Parameters
d (*np.ndarray*) – A numpy array of shape *(N, M)*.

Returns
A *(N, M)* array of hydrophoboic interactions in a piecewise linear curve.

Return type
np.ndarray

References

vina_hbond(*d: ndarray*) → ndarray

Computes Autodock Vina's hydrogen bond interaction term.

Here, d is the set of surface distances as defined in [1]_

Parameters
d (*np.ndarray*) – A numpy array of shape *(N, M)*.

Returns
A *(N, M)* array of hydrophoboic interactions in a piecewise linear curve.

Return type
np.ndarray

References

vina_gaussian_first(*d: ndarray*) → ndarray

Computes Autodock Vina's first Gaussian interaction term.

Here, d is the set of surface distances as defined in [1]_

Parameters
d (*np.ndarray*) – A numpy array of shape *(N, M)*.

Returns
A *(N, M)* array of gaussian interaction terms.

Return type
np.ndarray

References

vina_gaussian_second(*d: ndarray*) → ndarray

Computes Autodock Vina's second Gaussian interaction term.

Here, d is the set of surface distances as defined in [1]_

Parameters
d (*np.ndarray*) – A numpy array of shape *(N, M)*.

Returns
A *(N, M)* array of gaussian interaction terms.

Return type
np.ndarray

References

`vina_energy_term`(*coords1: ndarray*, *coords2: ndarray*, *weights: ndarray*, *wrot: float*, *Nrot: int*) → ndarray

Computes the Vina Energy function for two molecular conformations

Parameters

- **coords1** (*np.ndarray*) – Molecular coordinates of shape *(N, 3)*
- **coords2** (*np.ndarray*) – Molecular coordinates of shape *(M, 3)*
- **weights** (*np.ndarray*) – A numpy array of shape *(5,)*. The 5 values are weights for repulsion interaction term, hydrophobic interaction term, hydrogen bond interaction term, first Gaussian interaction term and second Gaussian interaction term.
- **wrot** (*float*) – The scaling factor for nonlinearity
- **Nrot** (*int*) – Number of rotatable bonds in this calculation

Returns
A scalar value with free energy

Return type
np.ndarray

3.31 Utilities

DeepChem has a broad collection of utility functions. Many of these maybe be of independent interest to users since they deal with some tricky aspects of processing scientific datatypes.

3.31.1 Data Utilities

Array Utilities

`pad_array`(*x: ndarray*, *shape: Tuple | int*, *fill: float = 0.0*, *both: bool = False*) → ndarray

Pad an array with a fill value.

Parameters

- **x** (*np.ndarray*) – A numpy array.
- **shape** (*Tuple or int*) – Desired shape. If int, all dimensions are padded to that size.
- **fill** (*float, optional (default 0.0)*) – The padded value.
- **both** (*bool, optional (default False)*) – If True, split the padding on both sides of each axis. If False, padding is applied to the end of each axis.

Returns
A padded numpy array

Return type
np.ndarray

Data Directory

The DeepChem data directory is where downloaded MoleculeNet datasets are stored.

get_data_dir() → str

Get the DeepChem data directory.

Returns
The default path to store DeepChem data. If you want to change this path, please set your own path to *DEEPCHEM_DATA_DIR* as an environment variable.

Return type
str

URL Handling

download_url(*url: str*, *dest_dir: str = '/tmp'*, *name: str | None = None*)

Download a file to disk.

Parameters

- **url** (*str*) – The URL to download from
- **dest_dir** (*str*) – The directory to save the file in
- **name** (*str*) – The file name to save it as. If omitted, it will try to extract a file name from the URL

File Handling

untargz_file(*file: str*, *dest_dir: str = '/tmp'*, *name: str | None = None*)

Untar and unzip a .tar.gz file to disk.

Parameters

- **file** (*str*) – The filepath to decompress
- **dest_dir** (*str*) – The directory to save the file in
- **name** (*str*) – The file name to save it as. If omitted, it will use the file name

unzip_file(*file: str*, *dest_dir: str = '/tmp'*, *name: str | None = None*)

Unzip a .zip file to disk.

Parameters

- **file** (*str*) – The filepath to decompress
- **dest_dir** (*str*) – The directory to save the file in
- **name** (*str*) – The directory name to unzip it to. If omitted, it will use the file name

load_data(*input_files: List[str]*, *shard_size: int | None = None*) → Iterator[Any]

Loads data from files.

Parameters

- **input_files** (*List[str]*) – List of filenames.
- **shard_size** (*int, default None*) – Size of shard to yield

Returns
Iterator which iterates over provided files.

Return type
Iterator[Any]

Notes

The supported file types are SDF, CSV and Pickle.

`load_sdf_files`(*input_files: List[str]*, *clean_mols: bool = True*, *tasks: List[str] = []*, *shard_size: int | None = None*) → Iterator[DataFrame]

Load SDF file into dataframe.

Parameters

- **input_files** (*List[str]*) – List of filenames
- **clean_mols** (*bool, default True*) – Whether to sanitize molecules.
- **tasks** (*List[str], default []*) – Each entry in *tasks* is treated as a property in the SDF file and is retrieved with *mol.GetProp(str(task))* where *mol* is the RDKit mol loaded from a given SDF entry.
- **shard_size** (*int, default None*) – The shard size to yield at one time.

Returns
Generator which yields the dataframe which is the same shard size.

Return type
Iterator[pd.DataFrame]

Notes

This function requires RDKit to be installed.

`load_csv_files`(*input_files: List[str]*, *shard_size: int | None = None*) → Iterator[DataFrame]

Load data as pandas dataframe from CSV files.

Parameters

- **input_files** (*List[str]*) – List of filenames
- **shard_size** (*int, default None*) – The shard size to yield at one time.

Returns
Generator which yields the dataframe which is the same shard size.

Return type
Iterator[pd.DataFrame]

`load_json_files`(*input_files: List[str]*, *shard_size: int | None = None*) → Iterator[DataFrame]

Load data as pandas dataframe.

Parameters

- **input_files** (*List[str]*) – List of json filenames.
- **shard_size** (*int, default None*) – Chunksize for reading json files.

Returns
Generator which yields the dataframe which is the same shard size.

Return type
Iterator[pd.DataFrame]

Notes

To load shards from a json file into a Pandas dataframe, the file must be originally saved with `df.to_json('filename.json', orient='records', lines=True)`

`load_pickle_files`(*input_files: List[str]*) → Iterator[Any]

Load dataset from pickle files.

Parameters
`input_files` (*List[str]*) – The list of filenames of pickle file. This function can load from gzipped pickle file like *XXXX.pkl.gz*.

Returns
Generator which yields the objects which is loaded from each pickle file.

Return type
Iterator[Any]

`load_from_disk`(*filename: str*) → Any

Load a dataset from file.

Parameters
`filename` (*str*) – A filename you want to load data.

Returns
A loaded object from file.

Return type
Any

`save_to_disk`(*dataset: Any*, *filename: str*, *compress: int = 3*)

Save a dataset to file.

Parameters

- `dataset` (*str*) – A data saved
- `filename` (*str*) – Path to save data.
- `compress` (*int, default 3*) – The compress option when dumping joblib file.

`load_dataset_from_disk`(*save_dir: str*) → Tuple[bool, Tuple[*DiskDataset*, *DiskDataset*, *DiskDataset*] | None, List[*Transformer*]]

Loads MoleculeNet train/valid/test/transformers from disk.

Expects that data was saved using *save_dataset_to_disk* below. Expects the following directory structure for *save_dir*: save_dir/

—> train_dir/ | —> valid_dir/ | —> test_dir/ | —> transformers.pkl

Parameters
`save_dir` (*str*) – Directory name to load datasets.

Returns

- **loaded** (*bool*) – Whether the load succeeded

- **all_dataset** (*Tuple[DiskDataset, DiskDataset, DiskDataset]*) – The train, valid, test datasets
- **transformers** (*Transformer*) – The transformers used for this dataset

See also:

save_dataset_to_disk

save_dataset_to_disk(*save_dir: str*, *train:* DiskDataset, *valid:* DiskDataset, *test:* DiskDataset, *transformers: List[*Transformer*]*)

Utility used by MoleculeNet to save train/valid/test datasets.

This utility function saves a train/valid/test split of a dataset along with transformers in the same directory. The saved datasets will take the following structure: save_dir/

—> train_dir/ | —> valid_dir/ | —> test_dir/ | —> transformers.pkl

Parameters

- **save_dir** (*str*) – Directory name to save datasets to.
- **train** (DiskDataset) – Training dataset to save.
- **valid** (DiskDataset) – Validation dataset to save.
- **test** (DiskDataset) – Test dataset to save.
- **transformers** (*List[*Transformer*]*) – List of transformers to save to disk.

See also:

load_dataset_from_disk

3.31.2 Molecular Utilities

class ConformerGenerator(*max_conformers: int = 1*, *rmsd_threshold: float = 0.5*, *force_field: str = 'uff'*, *pool_multiplier: int = 10*)

Generate molecule conformers.

Notes

Procedure 1. Generate a pool of conformers. 2. Minimize conformers. 3. Prune conformers using an RMSD threshold.

Note that pruning is done _after_ minimization, which differs from the protocol described in the references **[1]_ [2]_**.

References

Notes

This class requires RDKit to be installed.

`__init__`(*max_conformers: int = 1*, *rmsd_threshold: float = 0.5*, *force_field: str = 'uff'*, *pool_multiplier: int = 10*)

Parameters

- `max_conformers` (*int, optional (default 1)*) – Maximum number of conformers to generate (after pruning).
- `rmsd_threshold` (*float, optional (default 0.5)*) – RMSD threshold for pruning conformers. If None or negative, no pruning is performed.
- `force_field` (*str, optional (default 'uff')*) – Force field to use for conformer energy calculation and minimization. Options are 'uff', 'mmff94', and 'mmff94s'.
- `pool_multiplier` (*int, optional (default 10)*) – Factor to multiply by max_conformers to generate the initial conformer pool. Since conformers are pruned after energy minimization, increasing the size of the pool increases the chance of identifying max_conformers unique conformers.

`generate_conformers`(*mol: Any*) → Any

Generate conformers for a molecule.

This function returns a copy of the original molecule with embedded conformers.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
mol – A new RDKit Mol object containing the chosen conformers, sorted by increasing energy.

Return type
rdkit.Chem.rdchem.Mol

`embed_molecule`(*mol: Any*) → Any

Generate conformers, possibly with pruning.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
mol – RDKit Mol object with embedded multiple conformers.

Return type
rdkit.Chem.rdchem.Mol

`get_molecule_force_field`(*mol: Any*, *conf_id: int | None = None*, ***kwargs*) → Any

Get a force field for a molecule.

Parameters

- `mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object with embedded conformers.
- `conf_id` (*int, optional*) – ID of the conformer to associate with the force field.
- `kwargs` (*dict, optional*) – Keyword arguments for force field constructor.

Returns
ff – RDKit force field instance for a molecule.

Return type
rdkit.ForceField.rdForceField.ForceField

`minimize_conformers`(*mol: Any*) → None

Minimize molecule conformers.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object with embedded conformers.

`get_conformer_energies`(*mol: Any*) → ndarray

Calculate conformer energies.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object with embedded conformers.

Returns
energies – Minimized conformer energies.

Return type
np.ndarray

`prune_conformers`(*mol: Any*) → Any

Prune conformers from a molecule using an RMSD threshold, starting with the lowest energy conformer.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
new_mol – A new rdkit.Chem.rdchem.Mol containing the chosen conformers, sorted by increasing energy.

Return type
rdkit.Chem.rdchem.Mol

`static get_conformer_rmsd`(*mol: Any*) → ndarray

Calculate conformer-conformer RMSD.

Parameters
`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit Mol object

Returns
rmsd – A conformer-conformer RMSD value. The shape is *(NumConformers, NumConformers)*

Return type
np.ndarray

`class MoleculeLoadException`(**args*, ***kwargs*)

`__init__`(**args*, ***kwargs*)

`get_xyz_from_mol`(*mol*)

Extracts a numpy array of coordinates from a molecules.

Returns a *(N, 3)* numpy array of 3d coords of given rdkit molecule

Parameters
`mol` (*rdkit Molecule*) – Molecule to extract coordinates for

Return type
Numpy ndarray of shape *(N, 3)* where *N = mol.GetNumAtoms()*.

`add_hydrogens_to_mol`(*mol*, *is_protein=False*)

Add hydrogens to a molecule object

Parameters

- **mol** (*Rdkit Mol*) – Molecule to hydrogenate
- **is_protein** (*bool, optional (default False)*) – Whether this molecule is a protein.

Return type
Rdkit Mol

Note: This function requires RDKit and PDBFixer to be installed.

`compute_charges`(*mol*)

Attempt to compute Gasteiger Charges on Mol

This also has the side effect of calculating charges on mol. The mol passed into this function has to already have been sanitized

Parameters
mol (*rdkit molecule*) –

Return type
No return since updates in place.

Note: This function requires RDKit to be installed.

`load_molecule`(*molecule_file*, *add_hydrogens=True*, *calc_charges=True*, *sanitize=True*, *is_protein=False*)

Converts molecule file to (xyz-coords, obmol object)

Given molecule_file, returns a tuple of xyz coords of molecule and an rdkit object representing that molecule in that order *(xyz, rdkit_mol)*. This ordering convention is used in the code in a few places.

Parameters

- **molecule_file** (*str*) – filename for molecule
- **add_hydrogens** (*bool, optional (default True)*) – If True, add hydrogens via pdbfixer
- **calc_charges** (*bool, optional (default True)*) – If True, add charges via rdkit
- **sanitize** (*bool, optional (default False)*) – If True, sanitize molecules via rdkit
- **is_protein** (*bool, optional (default False)*) – If True`, this molecule is loaded as a protein. This flag will affect some of the cleanup procedures applied.

Returns

- *Tuple (xyz, mol) if file contains single molecule. Else returns a*
- *list of the tuples for the separate molecules in this list.*

Note: This function requires RDKit to be installed.

write_molecule(*mol*, *outfile*, *is_protein=False*)

Write molecule to a file

This function writes a representation of the provided molecule to the specified *outfile*. Doesn't return anything.

Parameters

- **mol** (*rdkit Mol*) – Molecule to write
- **outfile** (*str*) – Filename to write mol to
- **is_protein** (*bool, optional*) – Is this molecule a protein?

Note: This function requires RDKit to be installed.

Raises

ValueError – if *outfile* isn't of a supported format.:

3.31.3 Molecular Fragment Utilities

It's often convenient to manipulate subsets of a molecule. The `MolecularFragment` class aids in such manipulations.

class MolecularFragment(*atoms: Sequence[Any]*, *coords: ndarray*)

A class that represents a fragment of a molecule.

It's often convenient to represent a fragment of a molecule. For example, if two molecules form a molecular complex, it may be useful to create two fragments which represent the subsets of each molecule that's close to the other molecule (in the contact region).

Ideally, we'd be able to do this in RDKit direct, but manipulating molecular fragments doesn't seem to be supported functionality.

Examples

```
>>> import numpy as np
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("C")
>>> coords = np.array([[0.0, 0.0, 0.0]])
>>> atom = mol.GetAtoms()[0]
>>> fragment = MolecularFragment([atom], coords)
```

__init__(*atoms: Sequence[Any]*, *coords: ndarray*)

Initialize this object.

Parameters

- **atoms** (*Iterable[rdkit.Chem.rdchem.Atom]*) – Each entry in this list should be a RDKit Atom.
- **coords** (*np.ndarray*) – Array of locations for atoms of shape *(N, 3)* where *N == len(atoms)*.

`GetAtoms()` → List[*AtomShim*]

Returns the list of atoms

Returns
list of atoms in this fragment.

Return type
List[*AtomShim*]

`GetNumAtoms()` → int

Returns the number of atoms

Returns
Number of atoms in this fragment.

Return type
int

`GetCoords()` → ndarray

Returns 3D coordinates for this fragment as numpy array.

Returns
A numpy array of shape *(N, 3)* with coordinates for this fragment. Here, N is the number of atoms.

Return type
np.ndarray

`class AtomShim`(*atomic_num: int*, *partial_charge: float*, *atom_coords: ndarray*)

This is a shim object wrapping an atom.

We use this class instead of raw RDKit atoms since manipulating a large number of rdkit Atoms seems to result in segfaults. Wrapping the basic information in an AtomShim seems to avoid issues.

`__init__`(*atomic_num: int*, *partial_charge: float*, *atom_coords: ndarray*)

Initialize this object

Parameters

- **atomic_num** (*int*) – Atomic number for this atom.
- **partial_charge** (*float*) – The partial Gasteiger charge for this atom
- **atom_coords** (*np.ndarray*) – Of shape (3,) with the coordinates of this atom

`GetAtomicNum()` → int

Returns atomic number for this atom.

Returns
Atomic number for this atom.

Return type
int

`GetPartialCharge()` → float

Returns partial charge for this atom.

Returns
A partial Gasteiger charge for this atom.

Return type
float

`GetCoords()` → ndarray

Returns 3D coordinates for this atom as numpy array.

Returns
Numpy array of shape *(3,)* with coordinates for this atom.

Return type
np.ndarray

`strip_hydrogens`(*coords: ndarray*, *mol: Any* | MolecularFragment) → Tuple[ndarray, *MolecularFragment*]

Strip the hydrogens from input molecule

Parameters

- **coords** (*np.ndarray*) – The coords must be of shape (N, 3) and correspond to coordinates of mol.
- **mol** (*rdkit.Chem.rdchem.Mol or* MolecularFragment) – The molecule to strip

Returns
A tuple of *(coords, mol_frag)* where *coords* is a numpy array of coordinates with hydrogen coordinates. *mol_frag* is a *MolecularFragment*.

Return type
Tuple[np.ndarray, *MolecularFragment*]

Notes

This function requires RDKit to be installed.

`merge_molecular_fragments`(*molecules: List[*MolecularFragment*]*) → *MolecularFragment* | None

Helper method to merge two molecular fragments.

Parameters
molecules (*List[*MolecularFragment*]*) – List of *MolecularFragment* objects.

Returns
Returns a merged *MolecularFragment*

Return type
Optional[*MolecularFragment*]

`get_contact_atom_indices`(*fragments: List[Tuple[ndarray, Any]]*, *cutoff: float = 4.5*) → List[List[int]]

Compute that atoms close to contact region.

Molecular complexes can get very large. This can make it unwieldy to compute functions on them. To improve memory usage, it can be very useful to trim out atoms that aren't close to contact regions. This function computes pairwise distances between all pairs of molecules in the molecular complex. If an atom is within cutoff distance of any atom on another molecule in the complex, it is regarded as a contact atom. Otherwise it is trimmed.

Parameters

- **fragments** (*List[Tuple[np.ndarray, rdkit.Chem.rdchem.Mol]]*) – As returned by *rdkit_utils.load_complex*, a list of tuples of *(coords, mol)* where *coords* is a *(N_atoms, 3)* array and *mol* is the rdkit molecule object.
- **cutoff** (*float, optional (default 4.5)*) – The cutoff distance in angstroms.

Returns
A list of length *len(molecular_complex)*. Each entry in this list is a list of atom indices from that molecule which should be kept, in sorted order.

Return type
List[List[int]]

reduce_molecular_complex_to_contacts(*fragments: List[Tuple[ndarray, Any]]*, *cutoff: float = 4.5*) → List[Tuple[ndarray, *MolecularFragment*]]

Reduce a molecular complex to only those atoms near a contact.

Molecular complexes can get very large. This can make it unwieldy to compute functions on them. To improve memory usage, it can be very useful to trim out atoms that aren't close to contact regions. This function takes in a molecular complex and returns a new molecular complex representation that contains only contact atoms. The contact atoms are computed by calling *get_contact_atom_indices* under the hood.

Parameters

- **fragments** (*List[Tuple[np.ndarray, rdkit.Chem.rdchem.Mol]]*) – As returned by *rdkit_utils.load_complex*, a list of tuples of *(coords, mol)* where *coords* is a *(N_atoms, 3)* array and *mol* is the rdkit molecule object.
- **cutoff** (*float*) – The cutoff distance in angstroms.

Returns
A list of length *len(molecular_complex)*. Each entry in this list is a tuple of *(coords, MolecularFragment)*. The coords is stripped down to *(N_contact_atoms, 3)* where *N_contact_atoms* is the number of contact atoms for this complex. *MolecularFragment* is used since it's tricky to make a RDKit sub-molecule.

Return type
List[Tuple[np.ndarray, *MolecularFragment*]]

3.31.4 Coordinate Box Utilities

class CoordinateBox(*x_range: Tuple[float, float]*, *y_range: Tuple[float, float]*, *z_range: Tuple[float, float]*)

A coordinate box that represents a block in space.

Molecular complexes are typically represented with atoms as coordinate points. Each complex is naturally associated with a number of different box regions. For example, the bounding box is a box that contains all atoms in the molecular complex. A binding pocket box is a box that focuses in on a binding region of a protein to a ligand. A interface box is the region in which two proteins have a bulk interaction.

The *CoordinateBox* class is designed to represent such regions of space. It consists of the coordinates of the box, and the collection of atoms that live in this box alongside their coordinates.

__init__(*x_range: Tuple[float, float]*, *y_range: Tuple[float, float]*, *z_range: Tuple[float, float]*)

Initialize this box.

Parameters

- **x_range** (*Tuple[float, float]*) – A tuple of *(x_min, x_max)* with max and min x-coordinates.
- **y_range** (*Tuple[float, float]*) – A tuple of *(y_min, y_max)* with max and min y-coordinates.
- **z_range** (*Tuple[float, float]*) – A tuple of *(z_min, z_max)* with max and min z-coordinates.

Raises
ValueError –

`__contains__`(*point: Sequence[float]*) → bool

Check whether a point is in this box.

Parameters
: `point` (*Sequence[float]*) – 3-tuple or list of length 3 or np.ndarray of shape *(3,)*. The *(x, y, z)* coordinates of a point in space.

Returns
: *True* if *other* is contained in this box.

Return type
: bool

`center`() → Tuple[float, float, float]

Computes the center of this box.

Returns
: *(x, y, z)* the coordinates of the center of the box.

Return type
: Tuple[float, float, float]

Examples

```
>>> box = CoordinateBox((0, 1), (0, 1), (0, 1))
>>> box.center()
(0.5, 0.5, 0.5)
```

`volume`() → float

Computes and returns the volume of this box.

Returns
: The volume of this box. Can be 0 if box is empty

Return type
: float

Examples

```
>>> box = CoordinateBox((0, 1), (0, 1), (0, 1))
>>> box.volume()
1
```

`contains`(*other: CoordinateBox*) → bool

Test whether this box contains another.

This method checks whether *other* is contained in this box.

Parameters
: `other` (`CoordinateBox`) – The box to check is contained in this box.

Returns
: *True* if *other* is contained in this box.

Return type
: bool

Raises
`ValueError` –

`intersect_interval`(*interval1: Tuple[float, float]*, *interval2: Tuple[float, float]*) → Tuple[float, float]

Computes the intersection of two intervals.

Parameters

- **interval1** (*Tuple[float, float]*) – Should be *(x1_min, x1_max)*
- **interval2** (*Tuple[float, float]*) – Should be *(x2_min, x2_max)*

Returns
x_intersect – Should be the intersection. If the intersection is empty returns *(0, 0)* to represent the empty set. Otherwise is *(max(x1_min, x2_min), min(x1_max, x2_max))*.

Return type
Tuple[float, float]

`union`(*box1:* CoordinateBox, *box2:* CoordinateBox) → *CoordinateBox*

Merges provided boxes to find the smallest union box.

This method merges the two provided boxes.

Parameters

- **box1** (CoordinateBox) – First box to merge in
- **box2** (CoordinateBox) – Second box to merge into this box

Returns
Smallest *CoordinateBox* that contains both *box1* and *box2*

Return type
CoordinateBox

`merge_overlapping_boxes`(*boxes: List[*CoordinateBox*]*, *threshold: float = 0.8*) → List[*CoordinateBox*]

Merge boxes which have an overlap greater than threshold.

Parameters

- **boxes** (*list[*CoordinateBox*]*) – A list of *CoordinateBox* objects.
- **threshold** (*float, default 0.8*) – The volume fraction of the boxes that must overlap for them to be merged together.

Returns
List[CoordinateBox] of merged boxes. This list will have length less than or equal to the length of *boxes*.

Return type
List[*CoordinateBox*]

`get_face_boxes`(*coords: ndarray*, *pad: float = 5.0*) → List[*CoordinateBox*]

For each face of the convex hull, compute a coordinate box around it.

The convex hull of a macromolecule will have a series of triangular faces. For each such triangular face, we construct a bounding box around this triangle. Think of this box as attempting to capture some binding interaction region whose exterior is controlled by the box. Note that this box will likely be a crude approximation, but the advantage of this technique is that it only uses simple geometry to provide some basic biological insight into the molecule at hand.

The *pad* parameter is used to control the amount of padding around the face to be used for the coordinate box.

Parameters

- **coords** (*np.ndarray*) – A numpy array of shape *(N, 3)*. The coordinates of a molecule.
- **pad** (*float, optional (default 5.0)*) – The number of angstroms to pad.

Returns
boxes – List of *CoordinateBox*

Return type
List[*CoordinateBox*]

Examples

```
>>> coords = np.array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]])
>>> boxes = get_face_boxes(coords, pad=5)
```

3.31.5 Evaluation Utils

class Evaluator(*model*, *dataset: Dataset*, *transformers: List[Transformer]*)

Class that evaluates a model on a given dataset.

The evaluator class is used to evaluate a *dc.models.Model* class on a given *dc.data.Dataset* object. The evaluator is aware of *dc.trans.Transformer* objects so will automatically undo any transformations which have been applied.

Examples

Evaluators allow for a model to be evaluated directly on a Metric for *sklearn*. Let's do a bit of setup constructing our dataset and model.

```
>>> import deepchem as dc
>>> import numpy as np
>>> X = np.random.rand(10, 5)
>>> y = np.random.rand(10, 1)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> model = dc.models.MultitaskRegressor(1, 5)
>>> transformers = []
```

Then you can evaluate this model as follows >>> import sklearn >>> evaluator = Evaluator(model, dataset, transformers) >>> multitask_scores = evaluator.compute_model_performance(… sklearn.metrics.mean_absolute_error)

Evaluators can also be used with *dc.metrics.Metric* objects as well in case you want to customize your metric further.

```
>>> evaluator = Evaluator(model, dataset, transformers)
>>> metric = dc.metrics.Metric(dc.metrics.mae_score)
>>> multitask_scores = evaluator.compute_model_performance(metric)
```

__init__(*model*, *dataset: Dataset*, *transformers: List[Transformer]*)

Initialize this evaluator

Parameters

- **model** (`Model`) – Model to evaluate. Note that this must be a regression or classification model and not a generative model.
- **dataset** (`Dataset`) – Dataset object to evaluate *model* on.
- **transformers** (*List[*`Transformer`*]*) – List of *dc.trans.Transformer* objects. These transformations must have been applied to *dataset* previously. The dataset will be untransformed for metric evaluation.

output_statistics(*scores: Dict[str, float]*, *stats_out: str*)

Write computed stats to file.

Parameters

- **scores** (*dict*) – Dictionary mapping names of metrics to scores.
- **stats_out** (*str*) – Name of file to write scores to.

output_predictions(*y_preds: ndarray*, *csv_out: str*)

Writes predictions to file.

Writes predictions made on *self.dataset* to a specified file on disk. *self.dataset.ids* are used to format predictions.

Parameters

- **y_preds** (*np.ndarray*) – Predictions to output
- **csv_out** (*str*) – Name of file to write predictions to.

compute_model_performance(*metrics:* Metric | *Callable[[...], Any]* | *List[*Metric*]* | *List[Callable[[...], Any]]*, *csv_out: str* | *None = None*, *stats_out: str* | *None = None*, *per_task_metrics: bool = False*, *use_sample_weights: bool = False*, *n_classes: int = 2*) → Dict[str, float] | Tuple[Dict[str, float], Dict[str, float]]

Computes statistics of model on test data and saves results to csv.

Parameters

- **metrics** (*dc.metrics.Metric/list[dc.metrics.Metric]/function*) – The set of metrics provided. This class attempts to do some intelligent handling of input. If a single *dc.metrics.Metric* object is provided or a list is provided, it will evaluate *self.model* on these metrics. If a function is provided, it is assumed to be a metric function that this method will attempt to wrap in a *dc.metrics.Metric* object. A metric function must accept two arguments, *y_true, y_pred* both of which are *np.ndarray* objects and return a floating point score. The metric function may also accept a keyword argument *sample_weight* to account for per-sample weights.
- **csv_out** (*str, optional (DEPRECATED)*) – Filename to write CSV of model predictions.
- **stats_out** (*str, optional (DEPRECATED)*) – Filename to write computed statistics.
- **per_task_metrics** (*bool, optional*) – If true, return computed metric for each task on multitask dataset.
- **use_sample_weights** (*bool, optional (default False)*) – If set, use per-sample weights *w*.
- **n_classes** (*int, optional (default None)*) – If specified, will use *n_classes* as the number of unique classes in *self.dataset*. Note that this argument will be ignored for regression metrics.

Returns

- **multitask_scores** (*dict*) – Dictionary mapping names of metrics to metric scores.
- **all_task_scores** (*dict, optional*) – If *per_task_metrics == True*, then returns a second dictionary of scores for each task separately.

class GeneratorEvaluator(*model*, *generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[Transformer]*, *labels: List | None = None*, *weights: List | None = None*)

Evaluate models on a stream of data.

This class is a partner class to *Evaluator*. Instead of operating over datasets this class operates over a generator which yields batches of data to feed into provided model.

Examples

```
>>> import deepchem as dc
>>> import numpy as np
>>> X = np.random.rand(10, 5)
>>> y = np.random.rand(10, 1)
>>> dataset = dc.data.NumpyDataset(X, y)
>>> model = dc.models.MultitaskRegressor(1, 5)
>>> generator = model.default_generator(dataset, pad_batches=False)
>>> transformers = []
```

Then you can evaluate this model as follows

```
>>> import sklearn
>>> evaluator = GeneratorEvaluator(model, generator, transformers)
>>> multitask_scores = evaluator.compute_model_performance(
...     sklearn.metrics.mean_absolute_error)
```

Evaluators can also be used with *dc.metrics.Metric* objects as well in case you want to customize your metric further. (Note that a given generator can only be used once so we have to redefine the generator here.)

```
>>> generator = model.default_generator(dataset, pad_batches=False)
>>> evaluator = GeneratorEvaluator(model, generator, transformers)
>>> metric = dc.metrics.Metric(dc.metrics.mae_score)
>>> multitask_scores = evaluator.compute_model_performance(metric)
```

__init__(*model*, *generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[Transformer]*, *labels: List | None = None*, *weights: List | None = None*)

Parameters

- **model** (Model) – Model to evaluate.
- **generator** (*generator*) – Generator which yields batches to feed into the model. For a KerasModel, it should be a tuple of the form (inputs, labels, weights). The "correct" way to create this generator is to use *model.default_generator* as shown in the example above.
- **transformers** (*List[Transformer]*) – Tranformers to "undo" when applied to the models outputs
- **labels** (*list of Layer*) – layers which are keys in the generator to compare to outputs
- **weights** (*list of Layer*) – layers which are keys in the generator for weight matrices

compute_model_performance(*metrics: Metric | Callable[[...], Any] | List[Metric] | List[Callable[[...], Any]], per_task_metrics: bool = False, use_sample_weights: bool = False, n_classes: int = 2*) → Dict[str, float] | Tuple[Dict[str, float], Dict[str, float]]

Computes statistics of model on test data and saves results to csv.

Parameters

- **metrics** (`dc.metrics.Metric/list[dc.metrics.Metric]/function`) – The set of metrics provided. This class attempts to do some intelligent handling of input. If a single *dc.metrics.Metric* object is provided or a list is provided, it will evaluate *self.model* on these metrics. If a function is provided, it is assumed to be a metric function that this method will attempt to wrap in a *dc.metrics.Metric* object. A metric function must accept two arguments, *y_true, y_pred* both of which are *np.ndarray* objects and return a floating point score.
- **per_task_metrics** (`bool, optional`) – If true, return computed metric for each task on multitask dataset.
- **use_sample_weights** (`bool, optional (default False)`) – If set, use per-sample weights *w*.
- **n_classes** (`int, optional (default None)`) – If specified, will assume that all *metrics* are classification metrics and will use *n_classes* as the number of unique classes in *self.dataset*.

Returns

- **multitask_scores** (*dict*) – Dictionary mapping names of metrics to metric scores.
- **all_task_scores** (*dict, optional*) – If *per_task_metrics == True*, then returns a second dictionary of scores for each task separately.

relative_difference(*x: ndarray, y: ndarray*) → ndarray

Compute the relative difference between x and y

The two argument arrays must have the same shape.

Parameters

- **x** (`np.ndarray`) – First input array
- **y** (`np.ndarray`) – Second input array

Returns

z – We will have *z == np.abs(x-y) / np.abs(max(x, y))*.

Return type

np.ndarray

3.31.6 Genomic Utilities

seq_one_hot_encode(*sequences, letters: str = 'ATCGN'*) → ndarray

One hot encodes list of genomic sequences.

Sequences encoded have shape (N_sequences, N_letters, sequence_length, 1). These sequences will be processed as images with one color channel.

Parameters

- **sequences** (`np.ndarray or Iterator[Bio.SeqRecord]`) – Iterable object of genetic sequences

- **letters** (*str, optional (default "ATCGN")*) – String with the set of possible letters in the sequences.

Raises

`ValueError:` – If sequences are of different lengths.

Returns

A numpy array of shape *(N_sequences, N_letters, sequence_length, 1)*.

Return type

np.ndarray

encode_bio_sequence(*fname: str*, *file_type: str = 'fasta'*, *letters: str = 'ATCGN'*) → ndarray

Loads a sequence file and returns an array of one-hot sequences.

Parameters

- **fname** (*str*) – Filename of fasta file.
- **file_type** (*str, optional (default "fasta")*) – The type of file encoding to process, e.g. fasta or fastq, this is passed to Biopython.SeqIO.parse.
- **letters** (*str, optional (default "ATCGN")*) – The set of letters that the sequences consist of, e.g. ATCG.

Returns

A numpy array of shape *(N_sequences, N_letters, sequence_length, 1)*.

Return type

np.ndarray

Notes

This function requires BioPython to be installed.

hhblits(*dataset_path*, *database=None*, *data_dir=None*, *evalue=0.001*, *num_iterations=2*, *num_threads=4*)

Run hhblits multisequence alignment search on a dataset. This function requires the hhblits binary to be installed and in the path. This function also requires a Hidden Markov Model reference database to be provided. Both can be found here: https://github.com/soedinglab/hh-suite

The database should be in the deepchem data directory or specified as an argument. To set the deepchem data directory, run this command in your environment:

export DEEPCHEM_DATA_DIR=<path to data directory>

Parameters

- **dataset_path** (*str*) – Path to single sequence or multiple sequence alignment (MSA) dataset. Results will be saved in this directory.
- **database** (*str*) – Name of database to search against. Note this is not the path, but the name of the database.
- **data_dir** (*str*) – Path to database directory.
- **evalue** (*float*) – E-value cutoff.
- **num_iterations** (*int*) – Number of iterations.
- **num_threads** (*int*) – Number of threads.

Returns

- **results** (*.a3m file*) – MSA file containing the results of the hhblits search.

- **results** (*.hhr file*) – hhsuite results file containing the results of the hhblits search.

Examples

```
>>> from deepchem.utils.sequence_utils import hhblits
>>> msa_path = hhblits('test/data/example.fasta', database='example_db', data_dir=
↪'test/data/', evalue=0.001, num_iterations=2, num_threads=4)
```

`hhsearch`(*dataset_path*, *database=None*, *data_dir=None*, *evalue=0.001*, *num_iterations=2*, *num_threads=4*)

Run hhsearch multisequence alignment search on a dataset. This function requires the hhblits binary to be installed and in the path. This function also requires a Hidden Markov Model reference database to be provided. Both can be found here: https://github.com/soedinglab/hh-suite

The database should be in the deepchem data directory or specified as an argument. To set the deepchem data directory, run this command in your environment:

export DEEPCHEM_DATA_DIR=<path to data directory>

Examples

```
>>> from deepchem.utils.sequence_utils import hhsearch
>>> msa_path = hhsearch('test/data/example.fasta', database='example_db', data_dir=
↪'test/data/', evalue=0.001, num_iterations=2, num_threads=4)
```

Parameters

- **dataset_path** (*str*) – Path to multiple sequence alignment dataset. Results will be saved in this directory.
- **database** (*str*) – Name of database to search against. Note this is not the path, but the name of the database.
- **data_dir** (*str*) – Path to database directory.
- **evalue** (*float*) – E-value cutoff.
- **num_iterations** (*int*) – Number of iterations.
- **num_threads** (*int*) – Number of threads.

Returns

- **results** (*.a3m file*) – MSA file containing the results of the hhblits search.
- **results** (*.hhr file*) – hhsuite results file containing the results of the hhblits search.

`MSA_to_dataset`(*msa_path*)

Convert a multiple sequence alignment to a NumpyDataset object.

3.31.7 Geometry Utilities

`unit_vector`(*vector: ndarray*) → ndarray

Returns the unit vector of the vector.

Parameters
`vector` (`np.ndarray`) – A numpy array of shape *(3,)*, where *3* is (x,y,z).

Returns
A numpy array of shape *(3,)*. The unit vector of the input vector.

Return type
np.ndarray

`angle_between`(*vector_i: ndarray*, *vector_j: ndarray*) → float

Returns the angle in radians between vectors "vector_i" and "vector_j"

Note that this function always returns the smaller of the two angles between the vectors (value between 0 and pi).

Parameters

- `vector_i` (`np.ndarray`) – A numpy array of shape *(3,)*, where *3* is (x,y,z).
- `vector_j` (`np.ndarray`) – A numpy array of shape *(3,)*, where *3* is (x,y,z).

Returns
The angle in radians between the two vectors.

Return type
np.ndarray

Examples

```
>>> print("%0.06f" % angle_between((1, 0, 0), (0, 1, 0)))
1.570796
>>> print("%0.06f" % angle_between((1, 0, 0), (1, 0, 0)))
0.000000
>>> print("%0.06f" % angle_between((1, 0, 0), (-1, 0, 0)))
3.141593
```

`generate_random_unit_vector`() → ndarray

Generate a random unit vector on the sphere S^2.

Citation: http://mathworld.wolfram.com/SpherePointPicking.html

Pseudocode:

a. Choose random theta element [0, 2*pi]
b. Choose random z element [-1, 1]
c. Compute output vector u: (x,y,z) = (sqrt(1-z^2)*cos(theta), sqrt(1-z^2)*sin(theta),z)

Returns
u – A numpy array of shape *(3,)*. u is an unit vector

Return type
np.ndarray

`generate_random_rotation_matrix()` → ndarray

Generates a random rotation matrix.

1. Generate a random unit vector u, randomly sampled from the

   unit sphere (see function generate_random_unit_vector() for details)

2. Generate a second random unit vector v

   a. If absolute value of u dot v > 0.99, repeat.

      (This is important for numerical stability. Intuition: we want them to be as linearly independent as possible or else the orthogonalized version of v will be much shorter in magnitude compared to u. I assume in Stack they took this from Gram-Schmidt orthogonalization?)

   b. v" = v - (u dot v)*u, i.e. subtract out the component of

      v that's in u's direction

   c. normalize v" (this isn"t in Stack but I assume it must be

      done)

3. find w = u cross v"
4. u, v", and w will form the columns of a rotation matrix, R.

   The intuition is that u, v" and w are, respectively, what the standard basis vectors e1, e2, and e3 will be mapped to under the transformation.

Returns
R – A numpy array of shape *(3, 3)*. R is a rotation matrix.

Return type
np.ndarray

`is_angle_within_cutoff`(*vector_i: ndarray*, *vector_j: ndarray*, *angle_cutoff: float*) → bool

A utility function to compute whether two vectors are within a cutoff from 180 degrees apart.

Parameters

- **vector_i** (*np.ndarray*) – A numpy array of shape (3,)`, where *3* is (x,y,z).
- **vector_j** (*np.ndarray*) – A numpy array of shape *(3,)*, where *3* is (x,y,z).
- **cutoff** (*float*) – The deviation from 180 (in degrees)

Returns
Whether two vectors are within a cutoff from 180 degrees apart

Return type
bool

3.31.8 Graph Utilities

fourier_encode_dist(*x*, *num_encodings=4*, *include_self=True*)

Fourier encode the input tensor *x* based on the specified number of encodings.

This function applies a Fourier encoding to the input tensor *x* by dividing it by a range of scales (2^i for i in range(num_encodings)) and then concatenating the sine and cosine of the scaled values. Optionally, the original input tensor can be included in the output.

Parameters

- **x** (*torch.Tensor*) – Input tensor to be Fourier encoded.
- **num_encodings** (*int, optional, default=4*) – Number of Fourier encodings to apply.
- **include_self** (*bool, optional, default=True*) – Whether to include the original input tensor in the output.

Returns

Fourier encoded tensor.

Return type

torch.Tensor

Examples

```
>>> import torch
>>> x = torch.tensor([1.0, 2.0, 3.0])
>>> encoded_x = fourier_encode_dist(x, num_encodings=4, include_self=True)
```

aggregate_mean(*h*, ***kwargs*)

Compute the mean of the input tensor along the second to last dimension.

Parameters

h (*torch.Tensor*) – Input tensor.

Returns

Mean of the input tensor along the second to last dimension.

Return type

torch.Tensor

aggregate_max(*h*, ***kwargs*)

Compute the max of the input tensor along the second to last dimension.

Parameters

h (*torch.Tensor*) – Input tensor.

Returns

Max of the input tensor along the second to last dimension.

Return type

torch.Tensor

aggregate_min(*h*, ***kwargs*)

Compute the min of the input tensor along the second to last dimension.

Parameters

- **h** (*torch.Tensor*) – Input tensor.

- **`**kwargs`** – Additional keyword arguments.

Returns
Min of the input tensor along the second to last dimension.

Return type
torch.Tensor

`aggregate_std`(*h*, ***kwargs*)

Compute the standard deviation of the input tensor along the second to last dimension.

Parameters
h (*torch.Tensor*) – Input tensor.

Returns
Standard deviation of the input tensor along the second to last dimension.

Return type
torch.Tensor

`aggregate_var`(*h*, ***kwargs*)

Compute the variance of the input tensor along the second to last dimension.

Parameters
h (*torch.Tensor*) – Input tensor.

Returns
Variance of the input tensor along the second to last dimension.

Return type
torch.Tensor

`aggregate_moment`(*h*, *n=3*, ***kwargs*)

Compute the nth moment of the input tensor along the second to last dimension.

Parameters

- **h** (*torch.Tensor*) – Input tensor.
- **n** (*int, optional, default=3*) – The order of the moment to compute.

Returns
Nth moment of the input tensor along the second to last dimension.

Return type
torch.Tensor

`aggregate_sum`(*h*, ***kwargs*)

Compute the sum of the input tensor along the second to last dimension.

Parameters
h (*torch.Tensor*) – Input tensor.

Returns
Sum of the input tensor along the second to last dimension.

Return type
torch.Tensor

`scale_identity`(*h*, *D=None*, *avg_d=None*)

Identity scaling function.

Parameters

- **h** (*torch.Tensor*) – Input tensor.
- **D** (*torch.Tensor, optional*) – Degree tensor.
- **avg_d** (*dict, optional*) – Dictionary containing averages over the training set.

Returns
Scaled input tensor.

Return type
torch.Tensor

scale_amplification(*h*, *D*, *avg_d*)

Amplification scaling function. log(D + 1) / d * h where d is the average of the `log(D + 1)` in the training set

Parameters

- **h** (*torch.Tensor*) – Input tensor.
- **D** (*torch.Tensor*) – Degree tensor.
- **avg_d** (*dict*) – Dictionary containing averages over the training set.

Returns
Scaled input tensor.

Return type
torch.Tensor

scale_attenuation(*h*, *D*, *avg_d*)

Attenuation scaling function. (log(D + 1))^-1 / d * X where d is the average of the `log(D + 1))^-1` in the training set

Parameters

- **h** (*torch.Tensor*) – Input tensor.
- **D** (*torch.Tensor*) – Degree tensor.
- **avg_d** (*dict*) – Dictionary containing averages over the training set.

Returns
Scaled input tensor.

Return type
torch.Tensor

3.31.9 Hash Function Utilities

hash_ecfp(*ecfp: str*, *size: int = 1024*) → int

Returns an int < size representing given ECFP fragment.

Input must be a string. This utility function is used for various ECFP based fingerprints.

Parameters

- **ecfp** (*str*) – String to hash. Usually an ECFP fragment.
- **size** (*int, optional (default 1024)*) – Hash to an int in range [0, size)

Returns
ecfp_hash – An int < size representing given ECFP fragment

Return type
int

`hash_ecfp_pair`(*ecfp_pair: Tuple[str, str]*, *size: int = 1024*) → int

Returns an int < size representing that ECFP pair.

Input must be a tuple of strings. This utility is primarily used for spatial contact featurizers. For example, if a protein and ligand have close contact region, the first string could be the protein's fragment and the second the ligand's fragment. The pair could be hashed together to achieve one hash value for this contact region.

Parameters

- **ecfp_pair** (*Tuple[str, str]*) – Pair of ECFP fragment strings
- **size** (*int, optional (default 1024)*) – Hash to an int in range [0, size)

Returns
ecfp_hash – An int < size representing given ECFP pair.

Return type
int

`vectorize`(*hash_function: Callable[[Any, int], int]*, *feature_dict: Dict[int, str] | None = None*, *size: int = 1024*, *feature_list: List | None = None*) → ndarray

Helper function to vectorize a spatial description from a hash.

Hash functions are used to perform spatial featurizations in DeepChem. However, it's necessary to convert backwards from the hash function to feature vectors. This function aids in this conversion procedure. It creates a vector of zeros of length *size*. It then loops through *feature_dict*, uses *hash_function* to hash the stored value to an integer in range [0, size) and bumps that index.

Parameters

- **hash_function** (*Function, Callable[[str, int], int]*) – Should accept two arguments, *feature*, and *size* and return a hashed integer. Here *feature* is the item to hash, and *size* is an int. For example, if *size=1024*, then hashed values must fall in range *[0, 1024)*.
- **feature_dict** (*Dict, optional (default None)*) – Maps unique keys to features computed.
- **size** (*int (default 1024)*) – Length of generated bit vector
- **feature_list** (*List, optional (default None)*) – List of features.

Returns
feature_vector – A numpy array of shape *(size,)*

Return type
np.ndarray

3.31.10 Voxel Utils

`convert_atom_to_voxel`(*coordinates: ndarray*, *atom_index: int*, *box_width: float*, *voxel_width: float*) → ndarray

Converts atom coordinates to an i,j,k grid index.

This function offsets molecular atom coordinates by (box_width/2, box_width/2, box_width/2) and then divides by voxel_width to compute the voxel indices.

Parameters

- **coordinates** (*np.ndarray*) – Array with coordinates of all atoms in the molecule, shape (N, 3).

- **atom_index** (*int*) – Index of an atom in the molecule.
- **box_width** (*float*) – Size of the box in Angstroms.
- **voxel_width** (*float*) – Size of a voxel in Angstroms

Returns
indices – A 1D numpy array of length 3 with *[i, j, k]*, the voxel coordinates of specified atom.

Return type
np.ndarray

convert_atom_pair_to_voxel(*coordinates_tuple: Tuple[ndarray, ndarray]*, *atom_index_pair: Tuple[int, int]*, *box_width: float*, *voxel_width: float*) → ndarray

Converts a pair of atoms to i,j,k grid indexes.

Parameters

- **coordinates_tuple** (*Tuple[np.ndarray, np.ndarray]*) – A tuple containing two molecular coordinate arrays of shapes *(N, 3)* and *(M, 3)*.
- **atom_index_pair** (*Tuple[int, int]*) – A tuple of indices for the atoms in the two molecules.
- **box_width** (*float*) – Size of the box in Angstroms.
- **voxel_width** (*float*) – Size of a voxel in Angstroms

Returns
indices_list – A numpy array of shape *(2, 3)*, where *3* is *[i, j, k]* of the voxel coordinates of specified atom.

Return type
np.ndarray

voxelize(*get_voxels: Callable[[...], Any]*, *coordinates: Any*, *box_width: float = 16.0*, *voxel_width: float = 1.0*, *hash_function: Callable[[...], Any] | None = None*, *feature_dict: Dict[Any, Any] | None = None*, *feature_list: List[int | Tuple[int]] | None = None*, *nb_channel: int = 16*, *dtype: str = 'int'*) → ndarray

Helper function to voxelize inputs.

This helper function helps convert a hash function which specifies spatial features of a molecular complex into a voxel tensor. This utility is used by various featurizers that generate voxel grids.

Parameters

- **get_voxels** (*Function*) – Function that voxelizes inputs
- **coordinates** (*Any*) – Contains the 3D coordinates of a molecular system. This should have whatever type get_voxels() expects as its first argument.
- **box_width** (*float, optional (default 16.0)*) – Size of a box in which voxel features are calculated. Box is centered on a ligand centroid.
- **voxel_width** (*float, optional (default 1.0)*) – Size of a 3D voxel in a grid in Angstroms.
- **hash_function** (*Function*) – Used to map feature choices to voxel channels.
- **feature_dict** (*Dict, optional (default None)*) – Keys are atom indices or tuples of atom indices, the values are computed features. If *hash_function is not None*, then the values are hashed using the hash function into *[0, nb_channels)* and this channel at the voxel for the given key is incremented by *1* for each dictionary entry. If *hash_function is None*, then the value must be a vector of size *(n_channels,)* which is added to the existing channel values at that voxel grid.

- **feature_list** (*List, optional (default None)*) – List of atom indices or tuples of atom indices. This can only be used if *nb_channel==1*. Increments the voxels corresponding to these indices by *1* for each entry.
- **nb_channel** (*int, , optional (default 16)*) – The number of feature channels computed per voxel. Should be a power of 2.
- **dtype** (*str ('int' or 'float'), optional (default 'int')*) – The type of the numpy ndarray created to hold features.

Returns
feature_tensor – The voxel of the input with the shape *(voxels_per_edge, voxels_per_edge, voxels_per_edge, nb_channel)*.

Return type
np.ndarray

3.31.11 Graph Convolution Utilities

one_hot_encode(*val: int | str, allowable_set: List[str] | List[int], include_unknown_set: bool = False*) → List[float]

One hot encoder for elements of a provided set.

Examples

```
>>> one_hot_encode("a", ["a", "b", "c"])
[1.0, 0.0, 0.0]
>>> one_hot_encode(2, [0, 1, 2])
[0.0, 0.0, 1.0]
>>> one_hot_encode(3, [0, 1, 2])
[0.0, 0.0, 0.0]
>>> one_hot_encode(3, [0, 1, 2], True)
[0.0, 0.0, 0.0, 1.0]
```

Parameters

- **val** (*int or str*) – The value must be present in *allowable_set*.
- **allowable_set** (*List[int] or List[str]*) – List of allowable quantities.
- **include_unknown_set** (*bool, default False*) – If true, the index of all values not in *allowable_set* is *len(allowable_set)*.

Returns
An one-hot vector of val. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type
List[float]

Raises
ValueError – If include_unknown_set is False and *val* is not in *allowable_set*.

get_atom_type_one_hot(*atom: Any, allowable_set: List[str] = ['C', 'N', 'O', 'F', 'P', 'S', 'Cl', 'Br', 'I'], include_unknown_set: bool = True*) → List[float]

Get an one-hot feature of an atom type.

Parameters

- `atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object
- `allowable_set` (*List[str]*) – The atom types to consider. The default set is *["C", "N", "O", "F", "P", "S", "Cl", "Br", "I"].*
- `include_unknown_set` (*bool, default True*) – If true, the index of all atom not in *allowable_set* is *len(allowable_set).*

Returns

An one-hot vector of atom types. If *include_unknown_set* is False, the length is *len(allowable_set).* If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type

List[float]

`construct_hydrogen_bonding_info`(*mol: Any*) → List[Tuple[int, str]]

Construct hydrogen bonding infos about a molecule.

Parameters

`mol` (*rdkit.Chem.rdchem.Mol*) – RDKit mol object

Returns

A list of tuple *(atom_index, hydrogen_bonding_type).* The *hydrogen_bonding_type* value is "Acceptor" or "Donor".

Return type

List[Tuple[int, str]]

`get_atom_hydrogen_bonding_one_hot`(*atom: Any*, *hydrogen_bonding: List[Tuple[int, str]]*) → List[float]

Get an one-hot feat about whether an atom accepts electrons or donates electrons.

Parameters

- `atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object
- `hydrogen_bonding` (*List[Tuple[int, str]]*) – The return value of *construct_hydrogen_bonding_info*. The value is a list of tuple *(atom_index, hydrogen_bonding)* like (1, "Acceptor").

Returns

A one-hot vector of the ring size type. The first element indicates "Donor", and the second element indicates "Acceptor".

Return type

List[float]

`get_atom_is_in_aromatic_one_hot`(*atom: Any*) → List[float]

Get ans one-hot feature about whether an atom is in aromatic system or not.

Parameters

`atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object

Returns

A vector of whether an atom is in aromatic system or not.

Return type

List[float]

`get_atom_hybridization_one_hot`(*atom: Any*, *allowable_set: List[str] = ['SP', 'SP2', 'SP3']*, *include_unknown_set: bool = False*) → List[float]

Get an one-hot feature of hybridization type.

Parameters

- `atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object
- `allowable_set` (*List[str]*) – The hybridization types to consider. The default set is *["SP", "SP2", "SP3"]*
- `include_unknown_set` (*bool, default False*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns

An one-hot vector of the hybridization type. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type

List[float]

`get_atom_total_num_Hs_one_hot`(*atom: Any*, *allowable_set: List[int] = [0, 1, 2, 3, 4]*, *include_unknown_set: bool = True*) → List[float]

Get an one-hot feature of the number of hydrogens which an atom has.

Parameters

- `atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object
- `allowable_set` (*List[int]*) – The number of hydrogens to consider. The default set is *[0, 1, . . . , 4]*
- `include_unknown_set` (*bool, default True*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns

A one-hot vector of the number of hydrogens which an atom has. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type

List[float]

`get_atom_chirality_one_hot`(*atom: Any*) → List[float]

Get an one-hot feature about an atom chirality type.

Parameters

`atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object

Returns

A one-hot vector of the chirality type. The first element indicates "R", and the second element indicates "S".

Return type

List[float]

`get_atom_formal_charge`(*atom: Any*) → List[float]

Get a formal charge of an atom.

Parameters

`atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object

Returns

A vector of the formal charge.

Return type

List[float]

`get_atom_partial_charge`(*atom: Any*) → List[float]

Get a partial charge of an atom.

Parameters
`atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object

Returns
A vector of the parital charge.

Return type
List[float]

Notes

Before using this function, you must calculate *GasteigerCharge* like *AllChem.ComputeGasteigerCharges(mol)*.

`get_atom_total_degree_one_hot`(*atom: Any*, *allowable_set: List[int] = [0, 1, 2, 3, 4, 5]*, *include_unknown_set: bool = True*) → List[float]

Get an one-hot feature of the degree which an atom has.

Parameters

- `atom` (*rdkit.Chem.rdchem.Atom*) – RDKit atom object
- `allowable_set` (*List[int]*) – The degree to consider. The default set is *[0, 1, …, 5]*
- `include_unknown_set` (*bool, default True*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns
A one-hot vector of the degree which an atom has. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type
List[float]

`get_bond_type_one_hot`(*bond: Any*, *allowable_set: List[str] = ['SINGLE', 'DOUBLE', 'TRIPLE', 'AROMATIC']*, *include_unknown_set: bool = False*) → List[float]

Get an one-hot feature of bond type.

Parameters

- `bond` (*rdkit.Chem.rdchem.Bond*) – RDKit bond object
- `allowable_set` (*List[str]*) – The bond types to consider. The default set is *[“SINGLE”, “DOUBLE”, “TRIPLE”, “AROMATIC”]*.
- `include_unknown_set` (*bool, default False*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns
A one-hot vector of the bond type. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type
List[float]

`get_bond_is_in_same_ring_one_hot`(*bond: Any*) → List[float]

Get an one-hot feature about whether atoms of a bond is in the same ring or not.

Parameters
`bond` (*rdkit.Chem.rdchem.Bond*) – RDKit bond object

Returns
A one-hot vector of whether a bond is in the same ring or not.

Return type
List[float]

`get_bond_is_conjugated_one_hot`(*bond: Any*) → List[float]

Get an one-hot feature about whether a bond is conjugated or not.

Parameters
bond (*rdkit.Chem.rdchem.Bond*) – RDKit bond object

Returns
A one-hot vector of whether a bond is conjugated or not.

Return type
List[float]

`get_bond_stereo_one_hot`(*bond: Any*, *allowable_set: List[str] = ['STEREONONE', 'STEREOANY', 'STEREOZ', 'STEREOE']*, *include_unknown_set: bool = True*) → List[float]

Get an one-hot feature of the stereo configuration of a bond.

Parameters

- **bond** (*rdkit.Chem.rdchem.Bond*) – RDKit bond object
- **allowable_set** (*List[str]*) – The stereo configuration types to consider. The default set is *["STEREONONE", "STEREOANY", "STEREOZ", "STEREOE"]*.
- **include_unknown_set** (*bool, default True*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns
A one-hot vector of the stereo configuration of a bond. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type
List[float]

`get_bond_graph_distance_one_hot`(*bond: Any*, *graph_dist_matrix: ndarray*, *allowable_set: List[int] = [1, 2, 3, 4, 5, 6, 7]*, *include_unknown_set: bool = True*) → List[float]

Get an one-hot feature of graph distance.

Parameters

- **bond** (*rdkit.Chem.rdchem.Bond*) – RDKit bond object
- **graph_dist_matrix** (*np.ndarray*) – The return value of *Chem.GetDistanceMatrix(mol)*. The shape is *(num_atoms, num_atoms)*.
- **allowable_set** (*List[int]*) – The graph distance types to consider. The default set is *[1, 2, …, 7]*.
- **include_unknown_set** (*bool, default False*) – If true, the index of all types not in *allowable_set* is *len(allowable_set)*.

Returns
A one-hot vector of the graph distance. If *include_unknown_set* is False, the length is *len(allowable_set)*. If *include_unknown_set* is True, the length is *len(allowable_set) + 1*.

Return type
List[float]

3.31.12 Grover Utilities

extract_grover_attributes(*molgraph: BatchGraphData*)

Utility to extract grover attributes for grover model

Parameters

molgraph (*BatchGraphData*) – A batched graph data representing a collection of molecules.

Returns

graph_attributes – A tuple containing atom features, bond features, atom to bond mapping, bond to atom mapping, bond to reverse bond mapping, atom to atom mapping, atom scope, bond scope, functional group labels and other additional features.

Return type

Tuple

Example

```
>>> import deepchem as dc
>>> from deepchem.feat.graph_data import BatchGraphData
>>> smiles = ['CC', 'CCC', 'CC(=O)C']
>>> featurizer = dc.feat.GroverFeaturizer(features_generator=dc.feat.
↪CircularFingerprint())
>>> graphs = featurizer.featurize(smiles)
>>> molgraph = BatchGraphData(graphs)
>>> attributes = extract_grover_attributes(molgraph)
```

3.31.13 Debug Utilities

3.31.14 Docking Utilities

These utilities assist in file preparation and processing for molecular docking.

write_vina_conf(*protein_filename: str*, *ligand_filename: str*, *centroid: ndarray*, *box_dims: ndarray*, *conf_filename: str*, *num_modes: int = 9*, *exhaustiveness: int | None = None*) → None

Writes Vina configuration file to disk.

Autodock Vina accepts a configuration file which provides options under which Vina is invoked. This utility function writes a vina configuration file which directs Autodock vina to perform docking under the provided options.

Parameters

- **protein_filename** (*str*) – Filename for protein
- **ligand_filename** (*str*) – Filename for the ligand
- **centroid** (*np.ndarray*) – A numpy array with shape *(3,)* holding centroid of system
- **box_dims** (*np.ndarray*) – A numpy array of shape *(3,)* holding the size of the box to dock
- **conf_filename** (*str*) – Filename to write Autodock Vina configuration to.
- **num_modes** (*int, optional (default 9)*) – The number of binding modes Autodock Vina should find

- **exhaustiveness** (*int, optional*) – The exhaustiveness of the search to be performed by Vina

write_gnina_conf(*protein_filename: str*, *ligand_filename: str*, *conf_filename: str*, *num_modes: int = 9*, *exhaustiveness: int | None = None*, ***kwargs*) → None

Writes GNINA configuration file to disk.

GNINA accepts a configuration file which provides options under which GNINA is invoked. This utility function writes a configuration file which directs GNINA to perform docking under the provided options.

Parameters

- **protein_filename** (*str*) – Filename for protein
- **ligand_filename** (*str*) – Filename for the ligand
- **conf_filename** (*str*) – Filename to write Autodock Vina configuration to.
- **num_modes** (*int, optional (default 9)*) – The number of binding modes GNINA should find
- **exhaustiveness** (*int, optional*) – The exhaustiveness of the search to be performed by GNINA
- **kwargs** – Args supported by GNINA documented here https://github.com/gnina/gnina#usage

load_docked_ligands(*pdbqt_output: str*) → Tuple[List[Any], List[float]]

This function loads ligands docked by autodock vina.

Autodock vina writes outputs to disk in a PDBQT file format. This PDBQT file can contain multiple docked "poses". Recall that a pose is an energetically favorable 3D conformation of a molecule. This utility function reads and loads the structures for multiple poses from vina's output file.

Parameters

pdbqt_output (*str*) – Should be the filename of a file generated by autodock vina's docking software.

Returns

Tuple of *molecules, scores*. *molecules* is a list of rdkit molecules with 3D information. *scores* is the associated vina score.

Return type

Tuple[List[rdkit.Chem.rdchem.Mol], List[float]]

Notes

This function requires RDKit to be installed.

prepare_inputs(*protein: str*, *ligand: str*, *replace_nonstandard_residues: bool = True*, *remove_heterogens: bool = True*, *remove_water: bool = True*, *add_hydrogens: bool = True*, *pH: float = 7.0*, *optimize_ligand: bool = True*, *pdb_name: str | None = None*) → Tuple[Any, Any]

This prepares protein-ligand complexes for docking.

Autodock Vina requires PDB files for proteins and ligands with sensible inputs. This function uses PDBFixer and RDKit to ensure that inputs are reasonable and ready for docking. Default values are given for convenience, but fixing PDB files is complicated and human judgement is required to produce protein structures suitable for docking. Always inspect the results carefully before trying to perform docking.

Parameters

- **protein** (*str*) – Filename for protein PDB file or a PDBID.
- **ligand** (*str*) – Either a filename for a ligand PDB file or a SMILES string.
- **replace_nonstandard_residues** (*bool (default True)*) – Replace nonstandard residues with standard residues.
- **remove_heterogens** (*bool (default True)*) – Removes residues that are not standard amino acids or nucleotides.
- **remove_water** (*bool (default True)*) – Remove water molecules.
- **add_hydrogens** (*bool (default True)*) – Add missing hydrogens at the protonation state given by *pH*.
- **pH** (*float (default 7.0)*) – Most common form of each residue at given *pH* value is used.
- **optimize_ligand** (*bool (default True)*) – If True, optimize ligand with RDKit. Required for SMILES inputs.
- **pdb_name** (*Optional[str]*) – If given, write sanitized protein and ligand to files called "pdb_name.pdb" and "ligand_pdb_name.pdb"

Returns
Tuple of *protein_molecule, ligand_molecule* with 3D information.

Return type
Tuple[RDKitMol, RDKitMol]

Note: This function requires RDKit and OpenMM to be installed. Read more about PDBFixer here: https://github.com/openmm/pdbfixer.

Examples

```
>>> p, m = prepare_inputs('3cyx', 'CCC')
```

>> p.GetNumAtoms() >> m.GetNumAtoms()

```
>>> p, m = prepare_inputs('3cyx', 'CCC', remove_heterogens=False)
```

>> p.GetNumAtoms()

read_gnina_log(*log_file: str*) → ndarray

Read GNINA logfile and get docking scores.

GNINA writes computed binding affinities to a logfile.

Parameters
log_file (*str*) – Filename of logfile generated by GNINA.

Returns
scores – Array of binding affinity (kcal/mol), CNN pose score, and CNN affinity for each binding mode.

Return type
np.array, dimension (num_modes, 3)

Print Threshold

The printing threshold controls how many dataset elements are printed when `dc.data.Dataset` objects are converted to strings or represnted in the IPython repl.

get_print_threshold() → int

Return the printing threshold for datasets.

The print threshold is the number of elements from ids/tasks to print when printing representations of *Dataset* objects.

Returns
threshold – Number of elements that will be printed

Return type
int

set_print_threshold(*threshold: int*)

Set print threshold

The print threshold is the number of elements from ids/tasks to print when printing representations of *Dataset* objects.

Parameters
threshold (*int*) – Number of elements to print.

get_max_print_size() → int

Return the max print size for a dataset.

If a dataset is large, printing *self.ids* as part of a string representation can be very slow. This field controls the maximum size for a dataset before ids are no longer printed.

Returns
max_print_size – Maximum length of a dataset for ids to be printed in string representation.

Return type
int

set_max_print_size(*max_print_size: int*)

Set max_print_size

If a dataset is large, printing *self.ids* as part of a string representation can be very slow. This field controls the maximum size for a dataset before ids are no longer printed.

Parameters
max_print_size (*int*) – Maximum length of a dataset for ids to be printed in string representation.

3.31.15 Fake Data Generator

The utilities here are used to generate random sample data which can be used for testing model architectures or other purposes.

class **FakeGraphGenerator**(*min_nodes: int = 10*, *max_nodes: int = 10*, *n_node_features: int = 5*, *avg_degree: int = 4*, *n_edge_features: int = 3*, *n_classes: int = 2*, *task: str = 'graph'*, ***kwargs*)

Generates a random graphs which can be used for testing or other purposes.

The generated graph supports both node-level and graph-level labels.

Example

```
>>> from deepchem.utils.fake_data_generator import FakeGraphGenerator
>>> fgg  = FakeGraphGenerator(min_nodes=8, max_nodes=10,  n_node_features=5, avg_
↪degree=8, n_edge_features=3, n_classes=2, task='graph', z=5)
>>> graphs = fgg.sample(n_graphs=10)
>>> type(graphs)
<class 'deepchem.data.datasets.NumpyDataset'>
>>> type(graphs.X[0])
<class 'deepchem.feat.graph_data.GraphData'>
>>> len(graphs) == 10  # num_graphs
True
```

Note: The FakeGraphGenerator class is based on torch_geometric.dataset.FakeDataset class.

`__init__`(*min_nodes: int = 10*, *max_nodes: int = 10*, *n_node_features: int = 5*, *avg_degree: int = 4*, *n_edge_features: int = 3*, *n_classes: int = 2*, *task: str = 'graph'*, ***kwargs*)

Parameters

- **min_nodes** (*int, default 10*) – Minimum number of permissible nodes in a graph
- **max_nodes** (*int, default 10*) – Maximum number of permissible nodes in a graph
- **n_node_features** (*int, default 5*) – Average number of node features in a graph
- **avg_degree** (*int, default 4*) – Average degree of the graph (avg_degree should be a positive number greater than the min_nodes)
- **n_edge_features** (*int, default 3*) – Average number of features in the edge
- **task** (*str, default 'graph'*) – Indicates node-level labels or graph-level labels
- **kwargs** (*optional*) – Additional graph attributes and their shapes , e.g. *global_features = 5*

`sample`(*n_graphs: int = 100*) → *NumpyDataset*

Samples graphs

Parameters

n_graphs (*int, default 100*) – Number of graphs to generate

Returns

graphs – Generated Graphs

Return type

NumpyDataset

3.31.16 Electron Sampler

The utilities here are used to sample electrons in a given molecule and update it using monte carlo methods, which can be used for methods like Variational Monte Carlo, etc.

class ElectronSampler(*central_value: ndarray*, *f: Callable[[ndarray], ndarray]*, *batch_no: int = 10*, *x: ndarray = array([], dtype=float64)*, *steps: int = 200*, *steps_per_update: int = 10*, *seed: int | None = None*, *symmetric: bool = True*, *simultaneous: bool = True*)

This class enables to initialize electron's position using gauss distribution around a nucleus and update using Markov Chain Monte-Carlo(MCMC) moves.

Using the probability obtained from the square of magnitude of wavefunction of a molecule/atom, MCMC steps can be performed to get the electron's positions and further update the wavefunction. This method is primarily used in methods like Variational Monte Carlo to sample electrons around the nucleons. Sampling can be done in 2 ways: -Simultaneous: All the electrons' positions are updated all at once.

-Single-electron: MCMC steps are performed only a particular electron, given their index value.

Further these moves can be done in 2 methods: -Symmetric: In this configuration, the standard deviation for all the steps are uniform.

-Asymmetric: In this configuration, the standard deviation are not uniform and typically the standard deviation is obtained a function like harmonic distances, etc.

Irrespective of these methods, the initialization is done uniformly around the respective nucleus and the number of electrons specified.

Example

```
>>> from deepchem.utils.electron_sampler import ElectronSampler
>>> test_f = lambda x: 2*np.log(np.random.uniform(low=0,high=1.0,size=np.
↪shape(x)[0]))
>>> distribution=ElectronSampler(central_value=np.array([[1,1,3],[3,2,3]]),f=test_f,
↪seed=0,batch_no=2,steps=1000,)
>>> distribution.gauss_initialize_position(np.array([[1],[2]]))
```

>> print(distribution.x) [[[[1.03528105 1.00800314 3.01957476]]

[[3.01900177 1.99697286 2.99793562]]

[[3.00821197 2.00288087 3.02908547]]]

[[[1.04481786 1.03735116 2.98045444]]

[[3.01522075 2.0024335 3.00887726]]

[[3.00667349 2.02988158 2.99589683]]]]

```
>>> distribution.move()
0.5115
```

>> print(distribution.x) [[[[-0.32441754 1.23330263 2.67927645]]

[[3.42250997 2.23617126 3.55806632]]

[[3.37491385 1.54374006 3.13575241]]]

[[[0.49067726 1.03987841 3.70277884]]

[[3.5631939 1.68703947 2.5685874]]

[[2.84560249 1.73998364 3.41274181]]]]

`__init__`(*central_value: ndarray*, *f: Callable[[ndarray], ndarray]*, *batch_no: int = 10*, *x: ndarray = array([], dtype=float64)*, *steps: int = 200*, *steps_per_update: int = 10*, *seed: int | None = None*, *symmetric: bool = True*, *simultaneous: bool = True*)

Parameters

- `central_value` (*np.ndarray*) – Contains each nucleus' coordinates in a 2D array. The shape of the array should be(number_of_nucleus,3).Ex: [[1,2,3],[3,4,5],..]
- `f` (*Callable[[np.ndarray],np.ndarray]*) – A function that should give the twice the log probability of wavefunction of the molecular system when called. Should taken in a 4D array of electron's positions(x) as argument and return a numpy array containing the log probabilities of each batch.
- `batch_no` (*int, optional (default 10)*) – Number of batches of the electron's positions to be initialized.
- `x` (*np.ndarray, optional (default np.ndarray([]))*) – Contains the electron's coordinates in a 4D array. The shape of the array should be(batch_no,no_of_electrons,1,3). Can be a 1D empty array, when electron's positions are yet to be initialized.
- `steps` (*int, optional (default 10)*) – The number of MCMC steps to be performed when the moves are called.
- `steps_per_update` (*int (default 10)*) – The number of steps after which the parameters of the MCMC gets updated.
- `seed` (*int, optional (default None)*) – Random seed to use.
- `symmetric` (*bool, optional(default True)*) – If true, symmetric moves will be used, else asymmetric moves will be followed.
- `simultaneous` (*bool, optional(default True)*) – If true, MCMC steps will be performed on all the electrons, else only a single electron gets updated.

`sampled_electrons`

Keeps track of the sampled electrons at every step, must be empty at start.

Type

np.ndarray

`harmonic_mean`(*y: ndarray*) → ndarray

Calculates the harmonic mean of the value 'y' from the self.central value. The numpy array returned is typically scaled up to get the standard deviation matrix.

Parameters

`y` (*np.ndarray*) – Containing the data distribution. Shape of y should be (batch,no_of_electron,1,3)

Returns

Contains the harmonic mean of the data distribution of each batch. Shape of the array obtained (batch_no, no_of_electrons,1,1)

Return type

np.ndarray

`log_prob_gaussian`(*y: ndarray*, *mu: ndarray*, *sigma: ndarray*) → ndarray

Calculates the log probability of a gaussian distribution, given the mean and standard deviation

Parameters

- **y** (*np.ndarray*) – data for which the log normal distribution is to be found
- **mu** (*np.ndarray*) – Means wrt which the log normal is calculated. Same shape as x or should be brodcastable to x
- **sigma** (*np.ndarray,*) – The standard deviation of the log normal distribution. Same shape as x or should be brodcastable to x

Returns
Log probability of gaussian distribution, with the shape - (batch_no,).

Return type
np.ndarray

`gauss_initialize_position`(*no_sample: ndarray*, *stddev: float = 0.02*)

Initializes the position around a central value as mean sampled from a gauss distribution and updates self.x. :param no_sample: Contains the number of samples to initialize under each mean. should be in the form [[3],[2]..], where here it means 3 samples and 2 samples around the first entry and second entry,respectively in self.central_value is taken. :type no_sample: np.ndarray, :param stddev: contains the stddev with which the electrons' coordinates are initialized :type stddev: float, optional (default 0.02)

`electron_update`(*lp1*, *lp2*, *move_prob*, *ratio*, *x2*) → ndarray

Performs sampling & parameter updates of electrons and appends the sampled electrons to self.sampled_electrons.

Parameters

- **lp1** (*np.ndarray*) – Log probability of initial parameter state.
- **lp2** (*np.ndarray*) – Log probability of the new sampled state.
- **move_prob** (*np.ndarray*) – Sampled log probabilty of the electron moving from the initial to final state, sampled assymetrically or symetrically.
- **ratio** (*np.ndarray*) – Ratio of lp1 and lp2 state.
- **x2** (*np.ndarray*) – Numpy array of the new sampled electrons.

Returns
lp1 – The update log probability of initial parameter state.

Return type
np.ndarray

`move`(*stddev: float = 0.02*, *asymmetric_func: Callable[[ndarray], ndarray] | None = None*, *index: int | None = None*) → float

Performs Metropolis-Hasting move for self.x(electrons). The type of moves to be followed -(simultaneous or single-electron, symmetric or asymmetric) have been specified when calling the class. The self.x array is replaced with a new array at the end of each step containing the new electron's positions.

Parameters

- **asymmetric_func** (*Callable[[np.ndarray],np.ndarray], optional(default None)*) – Should be specified for an asymmetric move.The function should take in only 1 argument- y: a numpy array wrt to which mean should be calculated. This function should return the mean for the asymmetric proposal. For ferminet, this function is the harmonic mean of the distance between the electron and the nucleus.
- **stddev** (*float, optional (default 0.02)*) – Specifies the standard deviation in the case of symmetric moves and the scaling factor of the standard deviation matrix in the case of asymmetric moves.

- **index** (*int, optional (default None)*) – Specifies the index of the electron to be updated in the case of a single electron move.

Returns
accepted move ratio of the MCMC steps.

Return type
float

3.31.17 Density Functional Theory Utilities

The utilites here are used to create an object that contains information about a system's self-consistent iteration steps and other processes.

`class Lattice(a: Tensor)`

Lattice is an object that describe the periodicity of the lattice. Note that this object does not know about atoms. For the integrated object between the lattice and atoms, please see Sol

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import Lattice
>>> a = torch.tensor([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])
>>> lattice = Lattice(a)
>>> lattice.lattice_vectors()
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
>>> lattice.recip_vectors()
tensor([[6.2832, 0.0000, 0.0000],
        [0.0000, 6.2832, 0.0000],
        [0.0000, 0.0000, 6.2832]])
>>> lattice.volume() # volume of the unit cell
tensor(1.)
>>> lattice.get_lattice_ls(1.0) # get the neighboring lattice vectors
tensor([[ 0.,  0., -1.],
        [ 0., -1.,  0.],
        [-1.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> lattice.get_gvgrids(6.0) # get the neighboring G-vectors
(tensor([[ 0.0000,  0.0000, -6.2832],
        [ 0.0000, -6.2832,  0.0000],
        [-6.2832,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 6.2832,  0.0000,  0.0000],
        [ 0.0000,  6.2832,  0.0000],
        [ 0.0000,  0.0000,  6.2832]]), tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.
↪0000, 1.0000, 1.0000]))
>>> lattice.estimate_ewald_eta(1e-5) # estimate the ewald's sum eta
1.8
```

`__init__`(*a: Tensor*)

Initialize the lattice object.

2D or 1D repetition are not implemented yet

Parameters
`a` (*torch.Tensor*) – The lattice vectors with shape (ndim, ndim) with ndim == 3

`lattice_vectors`() → Tensor

Returns the 3D lattice vectors (nv, ndim) with nv == 3

`recip_vectors`() → Tensor

Returns the 3D reciprocal vectors with norm == 2 * pi with shape (nv, ndim) with nv == 3

Note: `torch.det(self.a)` should not be equal to zero.

`volume`() → Tensor

Returns the volume of a lattice.

`property params: Tuple[Tensor, ...]`

Returns the list of parameters of this object

`get_lattice_ls`(*rcut: float*, *exclude_zeros: bool = False*) → Tensor

Returns a tensor that contains the coordinates of the neighboring lattices.

Parameters

- **rcut** (*float*) – The threshold of the distance from the main cell to be included in the neighbor.
- **exclude_zeros** (*bool (default: False)*) – If True, then it will exclude the vector that are all zeros.

Returns
ls – Tensor with size *(nb, ndim)* containing the coordinates of the neighboring cells.

Return type
torch.Tensor

`get_gvgrids`(*gcut: float*, *exclude_zeros: bool = False*) → Tuple[Tensor, Tensor]

Returns a tensor that contains the coordinate in reciprocal space of the neighboring Brillouin zones.

Parameters

- **gcut** (*float*) – Cut off for generating the G-points.
- **exclude_zeros** (*bool (default: False)*) – If True, then it will exclude the vector that are all zeros.

Returns

- **gvgrids** (*torch.Tensor*) – Tensor with size *(ng, ndim)* containing the G-coordinates of the Brillouin zones.
- **weights** (*torch.Tensor*) – Tensor with size *(ng)* representing the weights of the G-points.

`estimate_ewald_eta`(*precision: float*) → float

estimate the ewald's sum eta for nuclei interaction energy the precision is assumed to be relative precision this formula is obtained by estimating the sum as an integral.

Parameters
precision (*float*) – The precision of the ewald's sum.

Returns
eta – The estimated eta.

Return type
float

`class SpinParam`(*u: T*, *d: T*)

Data structure to store different values for spin-up and spin-down electrons.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import SpinParam
>>> dens_u = torch.ones(1)
>>> dens_d = torch.zeros(1)
>>> sp = SpinParam(u=dens_u, d=dens_d)
>>> sp.u
tensor([1.])
>>> sp.sum()
tensor([1.])
>>> sp.reduce(torch.multiply)
tensor([0.])
```

`__init__`(*u: T*, *d: T*)

Initialize the SpinParam object.

Parameters

- **u** (*any type*) – The parameters that corresponds to the spin-up electrons.
- **d** (*any type*) – The parameters that corresponds to the spin-down electrons.

`sum`() → Any

Returns the sum of up and down parameters.

`reduce`(*fcn: Callable[[T, T], T]*) → T

Reduce up and down parameters with the given function.

`static apply_fcn`(*fcn: Callable[[...], P]*, **a*)

"Apply the function for each up and down elements of a

`class ValGrad`(*value: Tensor*, *grad: Tensor | None = None*, *lapl: Tensor | None = None*, *kin: Tensor | None = None*)

Data structure that contains local information about density profiles. Data structure used as a umbrella class for density profiles and the derivative of the potential w.r.t. density profiles.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad
>>> dens = torch.ones(1)
>>> grad = torch.zeros(1)
>>> lapl = torch.ones(1)
>>> kin = torch.ones(1)
>>> vg = ValGrad(value=dens, grad=grad, lapl=lapl, kin=kin)
>>> vg + vg
ValGrad(value=tensor([2.]), grad=tensor([0.]), lapl=tensor([2.]), kin=tensor([2.]))
>>> vg * 5
ValGrad(value=tensor([5.]), grad=tensor([0.]), lapl=tensor([5.]), kin=tensor([5.]))
```

`__init__`(*value: Tensor*, *grad: Tensor | None = None*, *lapl: Tensor | None = None*, *kin: Tensor | None = None*)

Initialize the ValGrad object.

Parameters

- **`value`** (*`torch.Tensor`*) – Tensors containing the value of the local information.
- **`grad`** (*`torch.Tensor or None`*) – If tensor, it represents the gradient of the local information with shape `(..., 3)` where `...` should be the same shape as `value`.
- **`lapl`** (*`torch.Tensor or None`*) – If tensor, represents the laplacian value of the local information. It should have the same shape as `value`.
- **`kin`** (*`torch.Tensor or None`*) – If tensor, represents the local kinetic energy density. It should have the same shape as `value`.

`__add__`(*b*)

Add two ValGrad objects together.

`__mul__`(*f: float | int | Tensor*)

Multiply the ValGrad object with a scalar.

`class` **`CGTOBasis`**(*angmom: int*, *alphas: Tensor*, *coeffs: Tensor*)

Data structure that contains information about a contracted gaussian type orbital (CGTO).

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import CGTOBasis
>>> alphas = torch.ones(1)
>>> coeffs = torch.ones(1)
>>> cgto = CGTOBasis(angmom=0, alphas=alphas, coeffs=coeffs)
>>> cgto.wfnormalize_()
CGTOBasis(angmom=0, alphas=tensor([1.]), coeffs=tensor([2.5265]), normalized=True)
```

`__init__`(*angmom: int*, *alphas: Tensor*, *coeffs: Tensor*)

Initialize the CGTOBasis object.

Parameters

- **`angmom`** (*`int`*) – The angular momentum of the basis.

- **alphas** (*torch.Tensor*) – The gaussian exponents of the basis. Shape: (nbasis,)
- **coeffs** (*torch.Tensor*) – The coefficients of the basis. Shape: (nbasis,)

wfnormalize_() → *CGTOBasis*

Wavefunction normalization

The normalization is obtained from CINTgto_norm from libcint/src/misc.c, or https://github.com/sunqm/libcint/blob/b8594f1d27c3dad9034984a2a5befb9d607d4932/src/misc.c#L80

Please note that the square of normalized wavefunctions do not integrate to 1, but e.g. for s: 4*pi, p: (4*pi/3)

class AtomCGTOBasis(*atomz: int | float | Tensor*, *bases: List[CGTOBasis]*, *pos: List[List[float]] | ndarray | Tensor*)

Data structure that contains information about a atom and its contracted gaussian type orbital (CGTO).

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import AtomCGTOBasis, CGTOBasis
>>> alphas = torch.ones(1)
>>> coeffs = torch.ones(1)
>>> cgto = CGTOBasis(angmom=0, alphas=alphas, coeffs=coeffs)
>>> atomcgto = AtomCGTOBasis(atomz=1, bases=[cgto], pos=[[0.0, 0.0, 0.0]])
>>> atomcgto
AtomCGTOBasis(atomz=1, bases=[CGTOBasis(angmom=0, alphas=tensor([1.]),␣
↪coeffs=tensor([1.]), normalized=False)], pos=tensor([[0., 0., 0.]]))
```

__init__(*atomz: int | float | Tensor*, *bases: List[CGTOBasis]*, *pos: List[List[float]] | ndarray | Tensor*)

Initialize the AtomCGTOBasis object.

Parameters

- **atomz** (*ZType*) – Atomic number of the atom.
- **bases** (*List[CGTOBasis]*) – List of CGTOBasis objects.
- **pos** (*AtomPosType*) – Position of the atom. Shape: (ndim,)

class BaseXC

This is the base class for the exchange-correlation (XC) functional. The XC functional is used to calculate the exchange-correlation energy and potential. The XC functional is usually divided into three families: LDA, GGA, and Meta-GGA. The LDA is the simplest one, which only depends on the density. The GGA depends on the density and its gradient. The Meta-GGA depends on the density, its gradient, and its Laplacian.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
```

(continues on next page)

(continued from previous page)

```
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                              d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None, lapl=None,␣
↪kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                      d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_fn=
↪<MulBackward0>), grad=None, lapl=None, kin=None))
```

abstract property family: int

Returns 1 for LDA, 2 for GGA, and 4 for Meta-GGA.

abstract get_edensityxc(*densinfo:* ValGrad | SpinParam*[*ValGrad*]*) → Tensor

Returns the xc energy density (energy per unit volume)

Parameters
densinfo (*Union[*ValGrad*,* SpinParam*[*ValGrad*]]*) – The density information. If the XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns
The energy density of the XC.

Return type
torch.Tensor

get_vxc(*densinfo:* ValGrad | SpinParam*[*ValGrad*]*)

Returns the ValGrad for the xc potential given the density info for unpolarized case.

This is the default implementation of vxc if there is no implementation in the specific class of XC.

Parameters
densinfo (*Union[*ValGrad*,* SpinParam*[*ValGrad*]]*) – The density information. If the

XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns
The ValGrad for the xc potential. If the XC is unpolarized, then the return is ValGrad. If the XC is polarized, then the return is SpinParam[ValGrad].

Return type
Union[*ValGrad*, *SpinParam*[*ValGrad*]]

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns
Sequence of name of parameters affecting the output of the method.

Return type
List[str]

Raises
KeyError – If the list in this function does not contain `methodname`.

__add__(*other: Any*) → Any

Add two BaseXC together

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC, AddBaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) -
↪> torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                              d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
```

(continues on next page)

(continued from previous page)

```
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                      d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc2 = AddBaseXC(xc, xc)
>>> xc2.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<AddBackward0>)
>>> xc2.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<AddBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<AddBackward0>), grad=None, lapl=None, kin=None))
>>> xc3 = xc + xc
>>> xc3.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<AddBackward0>)
>>> xc3.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<AddBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<AddBackward0>), grad=None, lapl=None, kin=None))
```

Parameters

other (BaseXC) – The BaseXC to be added with.

Returns

The BaseXC that is the sum of the two BaseXC.

Return type

BaseXC

__mul__(*other: float* | *int* | *Tensor*)

Multiply a BaseXC with a float or a tensor.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC, MulBaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) -
↪> torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                              d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                      d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc2 = MulBaseXC(xc, 2.)
>>> xc2.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc2.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc3 = xc * 2.
>>> xc3.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc3.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
```

Parameters
other (*Union[float, int, torch.Tensor]*) – The float or tensor to be multiplied with.

Returns
The BaseXC that is the product of the BaseXC and the float or tensor.

Return type
BaseXC

__rmul__(*other: float | int | Tensor*)
Multiply a BaseXC with a float or a tensor.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC, MulBaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) -
↪> torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                              d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                      d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc2 = MulBaseXC(xc, 2.)
>>> xc2.get_edensityxc(densinfo)
```

(continues on next page)

(continued from previous page)

```
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc2.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc3 = 2. * xc
>>> xc3.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc3.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_
↪fn=<MulBackward0>), grad=None, lapl=None, kin=None))
```

Parameters
: **other** (*Union[float, int, torch.Tensor]*) – The float or tensor to be multiplied with.

Returns
: The BaseXC that is the product of the BaseXC and the float or tensor.

Return type
: *BaseXC*

class AddBaseXC(*a:* BaseXC, *b:* BaseXC)

Add two BaseXC together

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC, AddBaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                              d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
```

(continues on next page)

(continued from previous page)

```
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None, lapl=None,␣
↪kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                        d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_fn=
↪<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc2 = AddBaseXC(xc, xc)
>>> xc2.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<AddBackward0>)
>>> xc2.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<AddBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_fn=
↪<AddBackward0>), grad=None, lapl=None, kin=None))
>>> xc3 = xc + xc
>>> xc3.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<AddBackward0>)
>>> xc3.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<AddBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_fn=
↪<AddBackward0>), grad=None, lapl=None, kin=None))
```

__init__(*a: BaseXC*, *b: BaseXC*) → None

Initialize the AddBaseXC

Parameters

- **a** (BaseXC) – BaseXC to be added to.
- **b** (BaseXC) – BaseXC to be added with.

property family

Returns 1 for LDA, 2 for GGA, and 4 for Meta-GGA.

get_vxc(*densinfo: ValGrad | SpinParam[ValGrad]*) → *ValGrad* | *SpinParam*[*ValGrad*]

Returns the ValGrad for the xc potential given the density info for unpolarized case.

Parameters

densinfo (*Union[ValGrad, SpinParam[ValGrad]]*) – The density information. If the XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns

The ValGrad for the xc potential. If the XC is unpolarized, then the return is ValGrad. If the XC is polarized, then the return is SpinParam[ValGrad].

Return type

Union[*ValGrad*, *SpinParam*[*ValGrad*]]

get_edensityxc(*densinfo: ValGrad | SpinParam[ValGrad]*) → Tensor

Returns the xc energy density (energy per unit volume)

Parameters

densinfo (*Union[ValGrad, SpinParam[ValGrad]]*) – The density information. If the XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns

The energy density of the XC.

Return type

torch.Tensor

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns

Sequence of name of parameters affecting the output of the method.

Return type

List[str]

Raises

KeyError – If the list in this function does not contain `methodname`.

class MulBaseXC(*a: BaseXC*, *b: float | Tensor*)

Multiply a BaseXC with a float or a tensor

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad, SpinParam
>>> from deepchem.utils.dft_utils import BaseXC, MulBaseXC
>>> class MyXC(BaseXC):
...     @property
...     def family(self) -> int:
...         return 1
...     def get_edensityxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪torch.Tensor:
...         if isinstance(densinfo, ValGrad):
...             return densinfo.value.pow(2)
...         else:
...             return densinfo.u.value.pow(2) + densinfo.d.value.pow(2)
...     def get_vxc(self, densinfo: Union[ValGrad, SpinParam[ValGrad]]) ->␣
↪Union[ValGrad, SpinParam[ValGrad]]:
...         if isinstance(densinfo, ValGrad):
...             return ValGrad(value=2*densinfo.value)
```

(continues on next page)

(continued from previous page)

```
...         else:
...             return SpinParam(u=ValGrad(value=2*densinfo.u.value),
...                             d=ValGrad(value=2*densinfo.d.value))
>>> xc = MyXC()
>>> densinfo = ValGrad(value=torch.tensor([1., 2., 3.], requires_grad=True))
>>> xc.get_edensityxc(densinfo)
tensor([1., 4., 9.], grad_fn=<PowBackward0>)
>>> xc.get_vxc(densinfo)
ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None, lapl=None,␣
↪kin=None)
>>> densinfo = SpinParam(u=ValGrad(value=torch.tensor([1., 2., 3.], requires_
↪grad=True)),
...                     d=ValGrad(value=torch.tensor([4., 5., 6.], requires_
↪grad=True)))
>>> xc.get_edensityxc(densinfo)
tensor([17., 29., 45.], grad_fn=<AddBackward0>)
>>> xc.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([2., 4., 6.], grad_fn=<MulBackward0>), grad=None,␣
↪lapl=None, kin=None), d=ValGrad(value=tensor([ 8., 10., 12.], grad_fn=
↪<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc2 = MulBaseXC(xc, 2.)
>>> xc2.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc2.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_fn=
↪<MulBackward0>), grad=None, lapl=None, kin=None))
>>> xc3 = xc * 2.
>>> xc3.get_edensityxc(densinfo)
tensor([34., 58., 90.], grad_fn=<MulBackward0>)
>>> xc3.get_vxc(densinfo)
SpinParam(u=ValGrad(value=tensor([ 4.,  8., 12.], grad_fn=<MulBackward0>),␣
↪grad=None, lapl=None, kin=None), d=ValGrad(value=tensor([16., 20., 24.], grad_fn=
↪<MulBackward0>), grad=None, lapl=None, kin=None))
```

`__init__`(*a:* BaseXC, *b: float* | *Tensor*) → None

Initialize the MulBaseXC

Parameters

- **a** (`BaseXC`) – BaseXC to be multiplied to.
- **b** (*`Union[float, torch.Tensor]`*) – float or tensor to be multiplied with.

`property family`

Returns 1 for LDA, 2 for GGA, and 4 for Meta-GGA.

`get_vxc`(*densinfo:* ValGrad | SpinParam*[*ValGrad*]*) → *ValGrad* | *SpinParam*[*ValGrad*]

Returns the ValGrad for the xc potential given the density info for unpolarized case.

Parameters

`densinfo` (*`Union[ValGrad, SpinParam[ValGrad]]`*) – The density information. If the XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns
The ValGrad for the xc potential. If the XC is unpolarized, then the return is ValGrad. If the XC is polarized, then the return is SpinParam[ValGrad].

Return type
Union[*ValGrad*, *SpinParam*[*ValGrad*]]

get_edensityxc(*densinfo: ValGrad | SpinParam[ValGrad]*) → Tensor

Returns the xc energy density (energy per unit volume)

Parameters
densinfo (*Union[ValGrad, SpinParam[ValGrad]]*) – The density information. If the XC is unpolarized, then densinfo is ValGrad. If the XC is polarized, then densinfo is SpinParam[ValGrad]. The ValGrad contains the value and gradient of the density. The SpinParam[ValGrad] contains the value and gradient of the density for each spin channel.

Returns
The energy density of the XC.

Return type
torch.Tensor

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns
Sequence of name of parameters affecting the output of the method.

Return type
List[str]

Raises
KeyError – If the list in this function does not contain `methodname`.

class CalcLDALibXCPol(**args*, ***kwargs*)

Local-density approximations (LDA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend solely upon the value of the electronic density at each point in space (and not, for example, derivatives of the density or the Kohn–Sham orbitals).

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("lda_x", "polarized")
>>> rho_u = torch.tensor([0.1, 0.2, 0.3])
>>> rho_d = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcLDALibXCPol.apply(rho_u, rho_d, 0, libxcfcn)[0]
>>> print(res)
tensor([[-0.0864, -0.2177, -0.3738]], dtype=torch.float64)
```

static forward(*ctx*, *rho_u: torch.Tensor*, *rho_d: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density for polarized LDA.

Parameters

- **rho_u** (*torch.Tensor*) – Density tensor for spin-up with shape (ninps)
- **rho_d** (*torch.Tensor*) – Density tensor for spin-down with shape (ninps)
- **deriv** (*int*) – Derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – libxc functional to use

Returns

Result is a tensor with shape (nderiv, ninps) where the first dimension indicates the result for derivatives of spin-up and spin-down and some of its combination.

Return type

Tuple[torch.Tensor]

static backward(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Calculates the gradient w.r.t. the input rho.

Parameters

grad_res (*torch.Tensor*) – Gradient of the result w.r.t. the result itself.

Returns

Gradient w.r.t. the input rho.

Return type

Tuple[torch.Tensor]

class CalcLDALibXCUnpol(**args*, ***kwargs*)

Calculates the energy density or its derivative w.r.t. density for unpolarized LDA.

Local-density approximations (LDA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend solely upon the value of the electronic density at each point in space (and not, for example, derivatives of the density or the Kohn–Sham orbitals).

The result is a tensor with shape (ninps).

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("lda_x", "unpolarized")
>>> rho = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcLDALibXCUnpol.apply(rho, 0, libxcfcn)[0]
>>> print(res)
tensor([[-0.0343, -0.0864, -0.1483]], dtype=torch.float64)
```

static forward(*ctx*, *rho: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density.

Parameters

- **rho** (*torch.Tensor*) – Density tensor with shape (ninps)

- **deriv** (*int*) – Derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – libxc functional to use

Returns
Result is a tensor with shape (ninps)

Return type
Tuple[torch.Tensor]

static backward(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Calculates the gradient w.r.t. the input rho.

Parameters
grad_res (*torch.Tensor*) – Gradient of the result w.r.t. the result itself.

Returns
Gradient w.r.t. the input rho.

Return type
Tuple[torch.Tensor]

class CalcGGALibXCUnpol(**args*, ***kwargs*)

Calculates the energy density or its derivative w.r.t. density for unpolarized GGA.

Generalized-gradient approximations (GGA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend not only upon the value of the electronic density at each point in space, but also upon its gradient.

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("gga_c_pbe", "unpolarized")
>>> rho = torch.tensor([0.1, 0.2, 0.3])
>>> sigma = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcGGALibXCUnpol.apply(rho, sigma, 0, libxcfcn)[0]
>>> print(res)
tensor([[-0.0016, -0.0070, -0.0137]], dtype=torch.float64)
```

static forward(*ctx*, *rho: torch.Tensor*, *sigma: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density and contracted gradient.

Every element in the tuple is a tensor with shape (ninps)

Parameters

- **rho** (*torch.Tensor*) – Density tensor with shape (ninps)
- **sigma** (*torch.Tensor*) – Contracted gradient tensor with shape (ninps)
- **deriv** (*int*) – Derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – libxc functional to use

static backward(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Calculates the gradient w.r.t. the input rho and sigma.

Parameters

grad_res (*torch.Tensor*) – Gradient of the result w.r.t. the result itself.

Returns

Gradient w.r.t. the input rho and sigma.

Return type

Tuple[torch.Tensor]

class CalcGGALibXCPol(**args*, ***kwargs*)

Calculates the energy density or its derivative w.r.t. density for polarized GGA.

Generalized-gradient approximations (GGA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend not only upon the value of the electronic density at each point in space, but also upon its gradient.

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("gga_c_pbe", "polarized")
>>> rho_u = torch.tensor([0.1, 0.2, 0.3])
>>> rho_d = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_uu = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_ud = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_dd = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcGGALibXCPol.apply(rho_u, rho_d, sigma_uu, sigma_ud, sigma_dd, 0,␣
↪libxcfcn)[0]
>>> print(res)
tensor([[-0.0047, -0.0175, -0.0322]], dtype=torch.float64)
```

static forward(*ctx*, *rho_u: torch.Tensor*, *rho_d: torch.Tensor*, *sigma_uu: torch.Tensor*, *sigma_ud: torch.Tensor*, *sigma_dd: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density and contracted gradient.

Parameters

- **rho_u** (*torch.Tensor*) – Density tensor for spin-up with shape (ninps)
- **rho_d** (*torch.Tensor*) – Density tensor for spin-down with shape (ninps)
- **sigma_uu** (*torch.Tensor*) – Contracted gradient tensor for spin-up with shape (ninps)
- **sigma_ud** (*torch.Tensor*) – Contracted gradient tensor for spin-up and spin-down with shape (ninps)
- **sigma_dd** (*torch.Tensor*) – Contracted gradient tensor for spin-down with shape (ninps)
- **deriv** (*int*) – Derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – libxc functional to use

Returns
Result is a tensor with shape (nderiv, ninps) where the first dimension indicates the result for derivatives of spin-up and spin-down and some of its combination.

Return type
Tuple[torch.Tensor]

static backward(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Returns the gradient w.r.t. the input rho and sigma.

Parameters
grad_res (*torch.Tensor*) – Gradient of the result w.r.t. the result itself.

Returns
Gradient w.r.t. the input rho and sigma.

Return type
Tuple[torch.Tensor]

class CalcMGGALibXCUnpol(**args*, ***kwargs*)

Calculates the energy density or its derivative w.r.t. density for unpolarized meta-GGA.

Meta-generalized-gradient approximations (meta-GGA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend not only upon the value of the electronic density at each point in space and its gradient, but also upon the Laplacian of the density and the kinetic energy density.

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("mgga_c_m06_l", "unpolarized")
>>> rho = torch.tensor([0.1, 0.2, 0.3])
>>> sigma = torch.tensor([0.1, 0.2, 0.3])
>>> lapl = torch.tensor([0.1, 0.2, 0.3])
>>> kin = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcMGGALibXCUnpol.apply(rho, sigma, lapl, kin, 0, libxcfcn)[0]
>>> print(res)
tensor([[-0.0032, -0.0066, -0.0087]], dtype=torch.float64)
```

static forward(*ctx*, *rho: torch.Tensor*, *sigma: torch.Tensor*, *lapl: torch.Tensor*, *kin: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density and contracted gradient.

Parameters

- **rho** (*torch.Tensor*) – Density tensor with shape (ninps)
- **sigma** (*torch.Tensor*) – Contracted gradient tensor with shape (ninps)
- **lapl** (*torch.Tensor*) – Laplacian tensor with shape (ninps)
- **kin** (*torch.Tensor*) – Kinetic energy density tensor with shape (ninps)
- **deriv** (*int*) – Derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – libxc functional to use

Returns
The result is a tensor with shape (nderiv, ninps)

Return type
Tuple[torch.Tensor]

`static backward`(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Returns the gradient w.r.t. the inputs

Parameters
grad_res (*torch.Tensor*) – The gradient of the result w.r.t. the result itself.

Returns
The gradient w.r.t. the inputs

Return type
Tuple[torch.Tensor]

`class CalcMGGALibXCPol`(**args*, ***kwargs*)

Calculates the energy density or its derivative w.r.t. density for polarized meta-GGA.

Meta-generalized-gradient approximations (meta-GGA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend not only upon the value of the electronic density at each point in space and its gradient, but also upon the Laplacian of the density and the kinetic energy density.

Examples

```
>>> import torch
>>> import pylibxc
>>> libxcfcn = pylibxc.LibXCFunctional("mgga_c_m06_l", "polarized")
>>> rho_u = torch.tensor([0.1, 0.2, 0.3])
>>> rho_d = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_uu = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_ud = torch.tensor([0.1, 0.2, 0.3])
>>> sigma_dd = torch.tensor([0.1, 0.2, 0.3])
>>> lapl_u = torch.tensor([0.1, 0.2, 0.3])
>>> lapl_d = torch.tensor([0.1, 0.2, 0.3])
>>> kin_u = torch.tensor([0.1, 0.2, 0.3])
>>> kin_d = torch.tensor([0.1, 0.2, 0.3])
>>> res = CalcMGGALibXCPol.apply(rho_u, rho_d, sigma_uu, sigma_ud, sigma_dd,
...                              lapl_u, lapl_d, kin_u, kin_d, 0, libxcfcn)[0]
>>> print(res)
tensor([[-0.0065, -0.0115, -0.0162]], dtype=torch.float64)
```

`static forward`(*ctx*, *rho_u: torch.Tensor*, *rho_d: torch.Tensor*, *sigma_uu: torch.Tensor*, *sigma_ud: torch.Tensor*, *sigma_dd: torch.Tensor*, *lapl_u: torch.Tensor*, *lapl_d: torch.Tensor*, *kin_u: torch.Tensor*, *kin_d: torch.Tensor*, *deriv: int*, *libxcfcn: pylibxc.functional.LibXCFunctional*) → Tuple[torch.Tensor, ...]

Calculates and returns the energy density or its derivative w.r.t. density and contracted gradient and laplacian and kinetic energy density. Every element in the tuple is a tensor with shape of (nderiv, ninps) where nderiv depends on the number of derivatives for spin-up and spin-down combinations, e.g. nderiv == 3 for vsigma (see libxc manual)

Parameters

- **rho_u** (*torch.Tensor*) – The density tensor for spin-up with shape (ninps)

- **rho_d** (*torch.Tensor*) – The density tensor for spin-down with shape (ninps)
- **sigma_uu** (*torch.Tensor*) – The contracted gradient tensor for spin-up with shape (ninps)
- **sigma_ud** (*torch.Tensor*) – The contracted gradient tensor for spin-up and spin-down with shape (ninps)
- **sigma_dd** (*torch.Tensor*) – The contracted gradient tensor for spin-down with shape (ninps)
- **lapl_u** (*torch.Tensor*) – The laplacian tensor for spin-up with shape (ninps)
- **lapl_d** (*torch.Tensor*) – The laplacian tensor for spin-down with shape (ninps)
- **kin_u** (*torch.Tensor*) – The kinetic energy density tensor for spin-up with shape (ninps)
- **kin_d** (*torch.Tensor*) – The kinetic energy density tensor for spin-down with shape (ninps)
- **deriv** (*int*) – The derivative order. 0 for energy density, 1 for derivative w.r.t. density, 2 for second derivative w.r.t. density, etc.
- **libxcfcn** (*pylibxc.functional.LibXCFunctional*) – The libxc functional to use

Returns
The result is a tensor with shape (nderiv, ninps) The result is a tensor with shape (nderiv, ninps) where the first dimension indicates the result for derivatives of spin-up and spin-down and some of its combination.

Return type
Tuple[torch.Tensor]

static backward(*ctx*, **grad_res: Tensor*) → Tuple[Tensor | None, ...]

Returns the gradient w.r.t. the input rho and sigma.

Parameters
grad_res (*torch.Tensor*) – The gradient of the result w.r.t. the result itself.

Returns
The gradient w.r.t. the input rho and sigma.

Return type
Tuple[torch.Tensor]

class LibXCLDA(*name: str*)

Local Density Approximation (LDA) wrapper for libxc. Local-density approximations (LDA) are a class of approximations to the exchange–correlation (XC) energy functional in density functional theory (DFT) that depend solely upon the value of the electronic density at each point in space.

Examples

```
>>> from deepchem.utils.dft_utils import ValGrad
>>> import torch
>>> # create a LDA wrapper for libxc
>>> lda = LibXCLDA("lda_x")
>>> # create a density information
>>> densinfo = ValGrad(value=torch.rand(2, 3, 4), grad=torch.rand(2, 3, 4, 3))
>>> # get the exchange-correlation potential
>>> potinfo = lda.get_vxc(densinfo)
```

(continues on next page)

(continued from previous page)

```
>>> potinfo.value.shape
torch.Size([2, 3, 4])
>>> edens = lda.get_edensityxc(densinfo)
>>> edens.shape
torch.Size([2, 3, 4])
```

_family

Family of the exchange-correlation functional

Type
int (default 1)

_unpolfcn_wrapper

Wrapper for the unpolarized LDA functional

Type
torch.autograd.Function (default CalcLDALibXCUnpol)

_polfcn_wrapper

Wrapper for the polarized LDA functional

Type
torch.autograd.Function (default CalcLDALibXCPol)

__init__(*name: str*) → None

Initialize the LDA wrapper for libxc.

Parameters
name (*str*) – Name of the exchange-correlation functional

property family: int

Get the family of the exchange-correlation functional.

Returns
Family of the exchange-correlation functional

Return type
int

get_vxc(*densinfo:* ValGrad | SpinParam*[*ValGrad*]*) → *ValGrad* | *SpinParam*[*ValGrad*]

Get the exchange-correlation potential from libxc.

Parameters
densinfo (*Union[ValGrad, SpinParam[ValGrad]]*) – density information densinfo.value: (*BD, nr) densinfo.grad: (*BD, nr, ndim)

Returns
Potential information potentialinfo.value: (*BD, nr) potentialinfo.grad: (*BD, nr, ndim)

Return type
Union[*ValGrad*, *SpinParam*[*ValGrad*]]

get_edensityxc(*densinfo:* ValGrad | SpinParam*[*ValGrad*]*) → Tensor

Get the exchange-correlation energy density from libxc.

Parameters
densinfo (*Union[ValGrad, SpinParam[ValGrad]]*) – Density information densinfo.value: (*BD, nr) densinfo.grad: (*BD, nr, ndim)

Returns
Exchange-correlation energy density edens: (*BD, nr)

Return type
torch.Tensor

`getparamnames`(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns
Sequence of name of parameters affecting the output of the method.

Return type
List[str]

Raises
KeyError – If the list in this function does not contain `methodname`.

`class LibXCGGA`(*name: str*)

Generalized Gradient Approximation (GGA) wrapper for libxc.

GGA can correct the overestimated binding energy of LDA in molecules and solids and extend the processing system to the energy and structure of the hydrogen bond system. The approximation greatly improves the calculation results of the energy related to electrons and exchange.

Examples

```
>>> from deepchem.utils.dft_utils import ValGrad
>>> import torch
>>> # create a GGA wrapper for libxc
>>> gga = LibXCGGA("gga_c_pbe")
>>> # create a density information
>>> n = 2
>>> rho_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> grad_u = torch.rand((3, n), dtype=torch.float64).requires_grad_()
>>> densinfo = ValGrad(value=rho_u, grad=grad_u)
>>> # get the exchange-correlation potential
>>> potinfo = gga.get_vxc(densinfo)
>>> potinfo.value.shape
torch.Size([2])
```

_family

Family of the exchange-correlation functional

Type
int (default 2)

_unpolfcn_wrapper

Wrapper for the unpolarized GGA functional

Type
torch.autograd.Function (default CalcGGALibXCUnpol)

_polfcn_wrapper
Wrapper for the polarized GGA functional

Type
torch.autograd.Function (default CalcGGALibXCPol)

__init__(*name: str*) → None
Initialize the LDA wrapper for libxc.

Parameters
name (*str*) – Name of the exchange-correlation functional

class LibXCMGGA(*name: str*)
Meta-Generalized Gradient Approximation (MGGA) wrapper for libxc.

Meta-GGAs typically improve upon the accuracy of GGAs as they additionally take into account the local kinetic energy density. This allows meta-GGAs to more accurately treat different chemical bonds (eg. covalent, metallic, and weak) compared to LDAs and GGAs.

Examples

```
>>> from deepchem.utils.dft_utils import ValGrad
>>> import torch
>>> # create a MGGA wrapper for libxc
>>> mgga = LibXCMGGA("mgga_x_scan")
>>> # create a density information
>>> n = 2
>>> rho_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> grad_u = torch.rand((3, n), dtype=torch.float64).requires_grad_()
>>> lapl_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> kin_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> densinfo = ValGrad(value=rho_u, grad=grad_u, lapl=lapl_u, kin=kin_u)
>>> # get the exchange-correlation potential
>>> potinfo = mgga.get_vxc(densinfo)
>>> potinfo.value.shape
torch.Size([2])
```

_family
Family of the exchange-correlation functional

Type
int (default 4)

_unpolfcn_wrapper
Wrapper for the unpolarized MGGA functional

Type
torch.autograd.Function (default CalcMGGALibXCUnpol)

_polfcn_wrapper
Wrapper for the polarized MGGA functional

Type
torch.autograd.Function (default CalcMGGALibXCPol)

__init__(*name: str*) → None

Initialize the LDA wrapper for libxc.

Parameters

name (*str*) – Name of the exchange-correlation functional

get_libxc(*name: str*) → *BaseXC*

Get the XC object of the libxc based on its libxc's name.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad
>>> xc = get_libxc("gga_c_pbe")
>>> n = 2
>>> rho_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> grad_u = torch.rand((3, n), dtype=torch.float64).requires_grad_()
>>> densinfo = ValGrad(value=rho_u, grad=grad_u)
>>> # get the exchange-correlation potential
>>> potinfo = xc.get_vxc(densinfo)
>>> potinfo.value.shape
torch.Size([2])
```

Parameters

name (*str*) – The full libxc name, e.g. "lda_c_pw"

Returns

XC object that wraps the xc requested

Return type

BaseXC

get_xc(*xcstr: str*) → *BaseXC*

Returns the XC object based on the expression in xcstr.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import ValGrad
>>> xc = get_xc("lda_x + gga_c_pbe")
>>> n = 2
>>> rho_u = torch.rand((n,), dtype=torch.float64).requires_grad_()
>>> grad_u = torch.rand((3, n), dtype=torch.float64).requires_grad_()
>>> densinfo = ValGrad(value=rho_u, grad=grad_u)
>>> # get the exchange-correlation potential
>>> potinfo = xc.get_vxc(densinfo)
>>> potinfo.value.shape
torch.Size([2])
```

Parameters

xcstr (*str*) – The expression of the xc string, e.g. `"lda_x + gga_c_pbe"` where the variable name will be replaced by the LibXC object

Returns
XC object based on the given expression

Return type
BaseXC

`class BaseGrid`

BaseGrid is a class that regulates the integration points over the spatial dimensions.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import BaseGrid
>>> class Grid(BaseGrid):
...     def __init__(self):
...         super(Grid, self).__init__()
...         self.ngrid = 10
...         self.ndim = 3
...         self.dvolume = torch.ones(self.ngrid, dtype=self.dtype, device=self.
↪device)
...         self.rgrid = torch.ones((self.ngrid, self.ndim), dtype=self.dtype,␣
↪device=self.device)
...     def get_dvolume(self):
...         return self.dvolume
...     def get_rgrid(self):
...         return self.rgrid
>>> grid = Grid()
>>> grid.get_dvolume()
tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> grid.get_rgrid()
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

References

Kasim, Muhammad F., and Sam M. Vinko. "Learning the exchange-correlation functional from nature with fully differentiable density functional theory." Physical Review Letters 127.12 (2021): 126403. https://github.com/diffqc/dqc/blob/0fe821fc92cb3457fb14f6dff0c223641c514ddb/dqc/grid/base_grid.py

`abstract property dtype: dtype`

dtype of the grid points.

Returns
dtype of the grid points

Return type
torch.dtype

`abstract property device: device`

device of the grid points

Returns
device of the grid points

Return type
torch.device

`abstract property coord_type: str`

type of the coordinate returned in get_rgrid. It can be 'cartesian' or 'spherical'.

Returns

- *str* – type of the coordinate returned in get_rgrid. It can be 'cartesian'
- *or 'spherical'.*

`abstract get_dvolume()` → Tensor

Obtain the torch.tensor containing the dV elements for the integration.

Returns
The dV elements for the integration. *BG is the length of the BaseGrid.

Return type
torch.tensor (*BG, ngrid)

`abstract get_rgrid()` → Tensor

Returns the grid points position in the specified coordinate in self.coord_type.

Returns
The grid points position. *BG is the length of the BaseGrid.

Return type
torch.tensor (*BG, ngrid, ndim)

`abstract getparamnames(`*methodname: str*, *prefix: str = ''*`)` → List[str]

Return a list with the parameter names corresponding to the given method (methodname)

Returns
List of parameter names of methodname

Return type
List[str]

`class BaseDF`

BaseDF represents the density fitting object used in calculating the electron repulsion (and xc energy) in Hamiltonian.

Density fitting in density functional theory (DFT) is a technique used to reduce the computational cost of evaluating electron repulsion integrals. In DFT, the key quantity is the electron density rather than the wave function, and the electron repulsion integrals involve four-electron interactions, making them computationally demanding.

Density fitting exploits the fact that many-electron integrals can be expressed as a sum of two-electron integrals involving auxiliary basis functions. By approximating these auxiliary basis functions, often referred to as fitting functions, the computational cost can be significantly reduced.

Examples

```
>>> from deepchem.utils.dft_utils import BaseDF
>>> import torch
>>> class MyDF(BaseDF):
...     def __init__(self):
...         super(MyDF, self).__init__()
...     def get_j2c(self):
...         return torch.ones((3, 3))
...     def get_j3c(self):
...         return torch.ones((3, 3, 3))
>>> df = MyDF()
>>> df.get_j2c()
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

abstract build() → *BaseDF*

Construct the matrices required to perform the calculation and return self.

Returns
The constructed density fitting object.

Return type
BaseDF

abstract get_elrep(*dm: Tensor*) → *LinearOperator*

Construct the electron repulsion linear operator from the given density matrix using the density fitting method.

Parameters
dm (*torch.Tensor*) – The density matrix.

Returns
The electron repulsion linear operator.

Return type
LinearOperator

abstract property j2c: Tensor

Returns the 2-centre 2-electron integrals of the auxiliary basis.

Returns
The 2-centre 2-electron integrals of the auxiliary basis.

Return type
torch.Tensor

abstract property j3c: Tensor

Return the 3-centre 2-electron integrals of the auxiliary basis and the basis.

Returns
The 3-centre 2-electron integrals of the auxiliary basis and the basis.

Return type
torch.Tensor

abstract getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str (default="")*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns

Sequence of name of parameters affecting the output of the method.

Return type

List[str]

class BaseHamilton

Hamilton is a class that provides the LinearOperator of the Hamiltonian components.

The Hamiltonian represents the total energy operator for a system of interacting electrons. The Kohn-Sham DFT approach introduces a set of fictitious non-interacting electrons that move in an effective potential. The total energy functional, which includes the kinetic energy of these fictitious electrons and their interaction with an effective potential (including the electron-electron interaction), is minimized to obtain the ground-state electronic structure.

The Kohn-Sham Hamiltonian is a key component of this approach, representing the operator that governs the evolution of the Kohn-Sham orbitals. It includes terms for the kinetic energy of electrons, the external potential (usually from nuclei), and the exchange-correlation potential that accounts for the electron-electron interactions.

The Fock matrix represents the one-electron part of the Hamiltonian matrix. Its components include kinetic energy, nuclear attraction, and electron-electron repulsion integrals. The Fock matrix is pivotal in solving the electronic Schrödinger equation and determining the electronic structure of molecular systems.

Examples

```
>>> from deepchem.utils.dft_utils import BaseHamilton
>>> class MyHamilton(BaseHamilton):
...     def __init__(self):
...         self._nao = 2
...         self._kpts = torch.tensor([[0.0, 0.0, 0.0]])
...         self._df = None
...     @property
...     def nao(self):
...         return self._nao
...     @property
...     def kpts(self):
...         return self._kpts
...     @property
...     def df(self):
...         return self._df
...     def build(self):
...         return self
...     def get_nuclattr(self):
...         return torch.ones((1, 1, self.nao, self.nao))
>>> ham = MyHamilton()
```

(continues on next page)

(continued from previous page)

```
>>> hamilton = ham.build()
>>> hamilton.get_nuclattr()
tensor([[[[[1., 1.],
          [1., 1.]]]]])
```

abstract property nao: int

Number of atomic orbital basis.

Returns
Number of atomic orbital basis.

Return type
int

abstract property kpts: Tensor

List of k-points in the Hamiltonian.

Returns
List of k-points in the Hamiltonian. Shape: (nkpts, ndim)

Return type
torch.Tensor

abstract property df: *BaseDF* | None

Returns the density fitting object (if any) attached to this Hamiltonian object.

Returns
Returns the density fitting object (if any) attached to this Hamiltonian object.

Return type
Optional[*BaseDF*]

abstract build()

Construct the elements needed for the Hamiltonian. Heavy-lifting operations should be put here.

abstract setup_grid(*grid:* BaseGrid, *xc:* BaseXC | *None* = *None*) → None

Setup the basis (with its grad) in the spatial grid and prepare the gradient of atomic orbital according to the ones required by the xc. If xc is not given, then only setup the grid with ao (without any gradients of ao)

Parameters

- **grid** (BaseGrid) – Grid used to setup this Hamilton.
- **xc** (*Optional[BaseXC] (default None)*) – Exchange Corelation functional of this Hamiltonian.

abstract get_nuclattr() → *LinearOperator*

LinearOperator of the nuclear Coulomb attraction.

Nuclear Coulomb attraction is the electrostatic force binding electrons to a nucleus. Positively charged protons attract negatively charged electrons, creating stability in quantum systems. This force plays a fundamental role in determining the structure and behavior of atoms, contributing significantly to the overall potential energy in atomic physics.

Returns
LinearOperator of the nuclear Coulomb attraction. Shape: (**BH*, nao, nao)

Return type
LinearOperator

abstract get_kinnucl() → *LinearOperator*

Returns the LinearOperator of the one-electron operator (i.e. kinetic and nuclear attraction). Action of a LinearOperator on a function is a linear transformation. In the case of one-electron operators, these transformations are essential for solving the Schrödinger equation and understanding the behavior of electrons in an atomic or molecular system.

Returns
LinearOperator of the one-electron operator. Shape: (**BH*, nao, nao)

Return type
LinearOperator

abstract get_overlap() → *LinearOperator*

Returns the LinearOperator representing the overlap of the basis. The overlap of the basis refers to the degree to which atomic or molecular orbitals in a quantum mechanical system share common space.

Returns
LinearOperator representing the overlap of the basis. Shape: (**BH*, nao, nao)

Return type
LinearOperator

abstract get_elrep(*dm: Tensor*) → *LinearOperator*

Obtains the LinearOperator of the Coulomb electron repulsion operator. Known as the J-matrix.

In the context of electronic structure theory, it accounts for the repulsive interaction between electrons in a many-electron system. The J-matrix elements involve the Coulombic interactions between pairs of electrons, influencing the total energy and behavior of the system.

Parameters
dm (*torch.Tensor*) – Density matrix. Shape: (**BD*, nao, nao)

Returns
LinearOperator of the Coulomb electron repulsion operator. Shape: (**BDH*, nao, nao)

Return type
LinearOperator

abstract get_exchange(*dm: Tensor* | SpinParam*[Tensor]*) → *LinearOperator* | *SpinParam*[*LinearOperator*]

Obtains the LinearOperator of the exchange operator. It is -0.5 * K where K is the K matrix obtained from 2-electron integral.

Exchange operator is a mathematical representation of the exchange interaction between identical particles, such as electrons. The exchange operator quantifies the effect of interchanging the positions of two particles.

Parameters
dm (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix. Shape: (**BD*, nao, nao)

Returns
LinearOperator of the exchange operator. Shape: (**BDH*, nao, nao)

Return type
Union[*LinearOperator*, *SpinParam*[*LinearOperator*]]

abstract get_vext(*vext: Tensor*) → *LinearOperator*

Returns a LinearOperator of the external potential in the grid.

$$\mathbf{V}_{ij} = \int b_i(\mathbf{r}) V(\mathbf{r}) b_j(\mathbf{r})\ d\mathbf{r}$$

External potential energy that a particle experiences in a discretized space or grid. In quantum mechanics or computational physics, when solving for the behavior of particles, an external potential is often introduced to represent the influence of external forces.

Parameters
`vext` (*torch.Tensor*) – External potential in the grid. Shape: (**BR*, ngrid)

Returns
LinearOperator of the external potential in the grid. Shape: (**BRH*, nao, nao)

Return type
LinearOperator

abstract get_vxc(*dm: Tensor* | SpinParam*[Tensor]*) → *LinearOperator* | *SpinParam*[*LinearOperator*]

Returns a LinearOperator for the exchange-correlation potential

The exchange-correlation potential combines two effects:

1. Exchange potential: Arises from the antisymmetry of the electron wave function. It quantifies the tendency of electrons to avoid each other due to their indistinguishability.

2. Correlation potential: Accounts for the electron-electron correlation effects that arise from the repulsion between electrons.

TODO: check if what we need for Meta-GGA involving kinetics and for exact-exchange

Parameters
`dm` (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix. Shape: (**BD*, nao, nao)

Returns
LinearOperator for the exchange-correlation potential. Shape: (**BDH*, nao, nao)

Return type
Union[*LinearOperator*, *SpinParam*[*LinearOperator*]]

abstract ao_orb2dm(*orb: Tensor*, *orb_weight: Tensor*) → Tensor

Convert the atomic orbital to the density matrix.

Parameters

- `orb` (*torch.Tensor*) – Atomic orbital. Shape: (*BO, nao, norb)
- `orb_weight` (*torch.Tensor*) – Orbital weight. Shape: (*BW, norb)

Returns
Density matrix. Shape: (*BOWH, nao, nao)

Return type
torch.Tensor

abstract aodm2dens(*dm: Tensor*, *xyz: Tensor*) → Tensor

Get the density value in the Cartesian coordinate.

Parameters

- `dm` (*torch.Tensor*) – Density matrix. Shape: (**BD*, nao, nao)
- `xyz` (*torch.Tensor*) – Cartesian coordinate. Shape: (**BR*, ndim)

Returns
Density value in the Cartesian coordinate. Shape: (**BRD*)

Return type
torch.Tensor

abstract get_e_hcore(*dm: Tensor*) → Tensor

Get the energy from the one-electron Hamiltonian. The input is total density matrix.

Parameters
`dm` (*torch.Tensor*) – Total Density matrix.

Returns
Energy from the one-electron Hamiltonian.

Return type
torch.Tensor

abstract get_e_elrep(*dm: Tensor*) → Tensor

Get the energy from the electron repulsion. The input is total density matrix.

Parameters
`dm` (*torch.Tensor*) – Total Density matrix.

Returns
Energy from the one-electron Hamiltonian.

Return type
torch.Tensor

abstract get_e_exchange(*dm: Tensor* | *SpinParam[Tensor]*) → Tensor

Get the energy from the exact exchange.

Parameters
`dm` (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix.

Returns
Energy from the exact exchange.

Return type
torch.Tensor

abstract get_e_xc(*dm: Tensor* | *SpinParam[Tensor]*) → Tensor

Returns the exchange-correlation energy using the xc object given in `.setup_grid()`

Parameters
`dm` (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix. Shape: (**BD*, nao, nao)

Returns
Exchange-correlation energy.

Return type
torch.Tensor

abstract ao_orb_params2dm(*ao_orb_params: Tensor*, *ao_orb_coeffs: Tensor*, *orb_weight: Tensor*, *with_penalty: float* | *None = None*) → List[Tensor]

Convert the atomic orbital free parameters (parametrized in such a way so it is not bounded) to the density matrix.

Parameters

- `ao_orb_params` (*torch.Tensor*) – The tensor that parametrized atomic orbital in an unbounded space.
- `ao_orb_coeffs` (*torch.Tensor*) – The tensor that helps `ao_orb_params` in describing the orbital. The difference with `ao_orb_params` is that `ao_orb_coeffs` is not differentiable and not to be optimized in variational method.

- **orb_weight** (*torch.Tensor*) – The orbital weights.
- **with_penalty** (*float or None*) – If a float, it returns a tuple of tensors where the first element is `dm`, and the second element is the penalty multiplied by the penalty weights. The penalty is to compensate the overparameterization of `ao_orb_params`, stabilizing the Hessian for gradient calculation.

Returns
The density matrix from the orbital parameters and (if `with_penalty`) the penalty of the overparameterization of `ao_orb_params`.

Return type
torch.Tensor or tuple of torch.Tensor

Notes

- The penalty should be 0 if `ao_orb_params` is from `dm2ao_orb_params`.
- The density matrix should be recoverable when put through `dm2ao_orb_params` and `ao_orb_params2dm`.

`abstract dm2ao_orb_params`(*dm: Tensor*, *norb: int*) → Tuple[Tensor, Tensor]

Convert from the density matrix to the orbital parameters. The map is not one-to-one, but instead one-to-many where there might be more than one orbital parameters to describe the same density matrix. For restricted systems, only one of the `dm` (`dm.u` or `dm.d`) is sufficient.

Parameters

- **dm** (*torch.Tensor*) – The density matrix.
- **norb** (*int*) – The number of orbitals for the system.

Returns
The atomic orbital parameters for the first returned value and the atomic orbital coefficients for the second value.

Return type
tuple of 2 torch.Tensor

`abstract getparamnames`(*methodname: str*, *prefix: str = ''*) → List[str]

Return the paramnames

Parameters

- **methodname** (*str*) – The name of the method.
- **prefix** (*str (default "")*) – The prefix of the paramnames.

Returns
The paramnames.

Return type
List[str]

`class _Config`(*THRESHOLD_MEMORY: int = 10737418240*, *CHUNK_MEMORY: int = 16777216*, *VERBOSE: int = 0*)

Contains the configuration for the DFT module

Examples

```
>>> from deepchem.utils.dft_utils.config import config
>>> Memory_usage = 1024**4 # Sample Memory usage by some Object/Matrix
>>> if Memory_usage > config.THRESHOLD_MEMORY :
...     print("Overload")
Overload
```

THRESHOLD_MEMORY

Threshold memory (matrix above this size should not be constructed)

Type
int (default=10*1024**3)

CHUNK_MEMORY

The memory for splitting big tensors into chunks.

Type
int (default=16*1024**2)

VERBOSE

Allowed Verbosity level (Defines the level of detail) Used by Looger for maintaining Logs.

Type
int (default=0)

Usage

-----

1. HamiltonCGTO

Type
Usage it for splitting big tensors into chunks.

__init__(*THRESHOLD_MEMORY: int = 10737418240*, *CHUNK_MEMORY: int = 16777216*, *VERBOSE: int = 0*) → None

class BaseOrbParams

Class that provides free-parameterization of orthogonal orbitals.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import BaseOrbParams
>>> class MyOrbParams(BaseOrbParams):
...     @staticmethod
...     def params2orb(params, coeffs, with_penalty):
...         return params, coeffs
...     @staticmethod
...     def orb2params(orb):
...         return orb, torch.tensor([0], dtype=orb.dtype, device=orb.device)
>>> params = torch.randn(3, 4, 5)
>>> coeffs = torch.randn(3, 4, 5)
>>> with_penalty = 0.1
```

(continues on next page)

(continued from previous page)

```
>>> orb, penalty = MyOrbParams.params2orb(params, coeffs, with_penalty)
>>> params2, coeffs2 = MyOrbParams.orb2params(orb)
>>> torch.allclose(params, params2)
True
```

static params2orb(*params: Tensor*, *coeffs: Tensor*, *with_penalty: float = 0.0*) → List[Tensor]

Convert the parameters & coefficients to the orthogonal orbitals. `params` is the tensor to be optimized in variational method, while `coeffs` is a tensor that is needed to get the orbital, but it is not optimized in the variational method.

Parameters

- **params** (*torch.Tensor*) – The free parameters to be optimized.
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals.
- **with_penalty** (*float (default 0.0)*) – If not 0.0, return the penalty term for the free parameters.

Returns

- **orb** (*torch.Tensor*) – The orthogonal orbitals.
- **penalty** (*torch.Tensor*) – The penalty term for the free parameters. If `with_penalty` is 0.0, this is not returned.

static orb2params(*orb: Tensor*) → List[Tensor]

Get the free parameters from the orthogonal orbitals. Returns `params` and `coeffs` described in `params2orb`.

Parameters

orb (*torch.Tensor*) – The orthogonal orbitals.

Returns

- **params** (*torch.Tensor*) – The free parameters to be optimized.
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals.

class QROrbParams

Orthogonal orbital parameterization using QR decomposition. The orthogonal orbital is represented by:

P = QR

Where Q is the parameters defining the rotation of the orthogonal tensor, and R is the coefficients tensor.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import QROrbParams
>>> params = torch.randn(3, 3)
>>> coeffs = torch.randn(4, 3)
>>> with_penalty = 0.1
>>> orb, penalty = QROrbParams.params2orb(params, coeffs, with_penalty)
>>> params2, coeffs2 = QROrbParams.orb2params(orb)
```

static params2orb(*params: Tensor*, *coeffs: Tensor*, *with_penalty: float = 0.0*) → List[Tensor]

Convert the parameters & coefficients to the orthogonal orbitals. `params` is the tensor to be optimized in variational method, while `coeffs` is a tensor that is needed to get the orbital, but it is not optimized in the variational method.

Parameters

- **params** (*torch.Tensor*) – The free parameters to be optimized.
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals.
- **with_penalty** (*float (default 0.0)*) – If not 0.0, return the penalty term for the free parameters.

Returns

- **orb** (*torch.Tensor*) – The orthogonal orbitals.
- **penalty** (*torch.Tensor*) – The penalty term for the free parameters. If `with_penalty` is 0.0, this is not returned.

static orb2params(*orb: Tensor*) → List[Tensor]

Get the free parameters from the orthogonal orbitals. Returns `params` and `coeffs` described in `params2orb`.

Parameters

orb (*torch.Tensor*) – The orthogonal orbitals.

Returns

- **params** (*torch.Tensor*) – The free parameters to be optimized.
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals.

class MatExpOrbParams

Orthogonal orbital parameterization using matrix exponential. The orthogonal orbital is represented by:

P = matrix_exp(Q) @ C

where C is an orthogonal coefficient tensor, and Q is the parameters defining the rotation of the orthogonal tensor.

Examples

```
>>> from deepchem.utils.dft_utils import MatExpOrbParams
>>> params = torch.randn(3, 3)
>>> coeffs = torch.randn(4, 3)
>>> with_penalty = 0.1
>>> orb, penalty = MatExpOrbParams.params2orb(params, coeffs, with_penalty)
>>> params2, coeffs2 = MatExpOrbParams.orb2params(orb)
```

static params2orb(*params: Tensor*, *coeffs: Tensor*, *with_penalty: float = 0.0*) → List[Tensor]

Convert the parameters & coefficients to the orthogonal orbitals. `params` is the tensor to be optimized in variational method, while `coeffs` is a tensor that is needed to get the orbital, but it is not optimized in the variational method.

Parameters

- **params** (*torch.Tensor*) – The free parameters to be optimized. (*, nparams)
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals. (*, nao, norb)

- **with_penalty** (*float (default 0.0)*) – If not 0.0, return the penalty term for the free parameters.

Returns

- **orb** (*torch.Tensor*) – The orthogonal orbitals.
- **penalty** (*torch.Tensor*) – The penalty term for the free parameters. If `with_penalty` is 0.0, this is not returned.

static orb2params(*orb: Tensor*) → List[Tensor]

Get the free parameters from the orthogonal orbitals. Returns `params` and `coeffs` described in `params2orb`.

Parameters

orb (*torch.Tensor*) – The orthogonal orbitals.

Returns

- **params** (*torch.Tensor*) – The free parameters to be optimized.
- **coeffs** (*torch.Tensor*) – The coefficients to get the orthogonal orbitals.

class parse_moldesc(*moldesc: str | Tuple[List[str] | List[int | float | Tensor] | Tensor, List[List[float]] | ndarray | Tensor], dtype: dtype = torch.float64, device: device = device(type='cpu')*)

Parse the string of molecular descriptor and returns tensors of atomzs and atom positions. .. rubric:: Examples

```
>>> from deepchem.utils.dft_utils import parse_moldesc
>>> system = {
...     'type': 'mol',
...     'kwargs': {
...         'moldesc': 'H 0.86625 0 0; F -0.86625 0 0',
...         'basis': '6-311++G(3df,3pd)'
...     }
... }
>>> atomzs, atomposs = parse_moldesc(system["kwargs"]["moldesc"])
>>> atomzs
tensor([1., 9.], dtype=torch.float64)
>>> atomposs
tensor([[ 0.8662,  0.0000,  0.0000],
        [-0.8662,  0.0000,  0.0000]], dtype=torch.float64)
```

Parameters

- **moldesc** (*Union[str, Tuple[AtomZsType, AtomPosType]]*) – String that describes the system, e.g. `"H -1 0 0; H 1 0 0"` for H2 molecule separated by 2 Bohr.
- **dtype** (*torch.dtype (default torch.float64)*) – The datatype of the returned atomic positions.
- **device** (*torch.device (default torch.device('cpu'))*) – The device to store the returned tensors.

Returns

- **atomzs** (*torch.Tensor*) – The tensor of atomzs [Atom Number].
- **atompos** (*torch.Tensor*) – The tensor of atomic positions [Bohr].

`class BaseSystem`

System is a class describing the environment before doing the quantum chemistry calculation. It contains the information of the atoms, the external electric field, the spin, the charge, etc. It also contains the Hamiltonian object and the grid object for the calculation. The system object is also responsible for setting up the cache for the parameters that can be read/written from/to the cache file.

Examples

```
>>> from deepchem.utils.dft_utils import BaseSystem
>>> from deepchem.utils.dft_utils import BaseHamilton
>>> from deepchem.utils.dft_utils import BaseGrid
>>> class MySystem(BaseSystem):
...     def __init__(self):
...         self.hamiltonian = BaseHamilton()
...         self.grid = BaseGrid()
...     def get_hamiltonian(self):
...         return self.hamiltonian
...     def get_grid(self):
...         return self.grid
...     def requires_grid(self):
...         return True
>>> system = MySystem()
>>> system.requires_grid()
True
```

abstract densityfit(*method: str | None = None, auxbasis: str | List[CGTOBasis] | List[str] | List[List[CGTOBasis]] | Dict[str | int, List[CGTOBasis] | str] | None = None*) → *BaseSystem*

Indicate that the system's Hamiltonian will use density fitting.

Parameters

- **method** (*Optional[str] (default None)*) – The density fitting method to use.
- **auxbasis** (*Optional[BasisInpType] (default None)*) – Auxiliary basis set to use for density fitting.

Returns

The system with density fitting enabled.

Return type

BaseSystem

abstract get_hamiltonian() → *BaseHamilton*

Hamiltonian object for the system.

Returns

Hamiltonian object for the system.

Return type

BaseHamilton

abstract set_cache(*fname: str, paramnames: List[str] | None = None*) → *BaseSystem*

Set up the cache to read/write some parameters from the given files. If paramnames is not given, then read/write all cache-able parameters specified by each class.

Parameters

- **fname** (*str*) – The file name of the cache file.
- **paramnames** (*Optional[List[str]] (default None)*) – The list of parameter names to read/write from the cache file.

Returns
The system with cache enabled.

Return type
BaseSystem

abstract get_orbweight(*polarized: bool = False*) → Tensor | *SpinParam*[Tensor]

Returns the atomic orbital weights. If polarized == False, then it returns the total orbital weights. Otherwise, it returns a tuple of orbital weights for spin-up and spin-down.

Parameters
polarized (*bool (default False)*) – Whether to return the orbital weights for each spin.

Returns
The orbital weights. Shape (*BS, norb)

Return type
Union[torch.Tensor, *SpinParam*[torch.Tensor]]

abstract get_nuclei_energy() → Tensor

Returns the nuclei-nuclei repulsion energy.

Returns
The nuclei-nuclei repulsion energy.

Return type
torch.Tensor

abstract setup_grid() → None

Construct the integration grid for the system.

abstract get_grid() → *BaseGrid*

Returns the grid of the system

abstract requires_grid() → bool

True if the system needs the grid to be constructed. Otherwise, returns False

abstract getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

Return a list with the parameter names corresponding to the given method (methodname)

Parameters

- **methodname** (*str*) – The name of the method.
- **prefix** (*str (default "")*) – The prefix of the parameter names.

Returns
List of parameter names of methodname

Return type
List[str]

abstract make_copy(***kwargs*) → *BaseSystem*

Copy of the system identical to the orginal except for new parameters set in the kwargs.

Parameters
kwargs – New parameters to set in the copy.

Returns
Copy of the system identical to the orginal except for new parameters set in the kwargs.

Return type
BaseSystem

abstract property atompos: Tensor

Atom positions with shape `(natoms, ndim)`.

Returns
Atom positions with shape `(natoms, ndim)`.

Return type
torch.Tensor

abstract property atomzs: Tensor

Atomic number with shape `(natoms,)`.

Returns
Atomic number with shape `(natoms,)`.

Return type
torch.Tensor

abstract property atommasses: Tensor

Atomic mass with shape `(natoms)` in atomic unit.

Returns
Atomic mass with shape `(natoms)` in atomic unit.

Return type
torch.Tensor

abstract property spin: int | float | Tensor

Total spin of the system.

Returns
Total spin of the system.

Return type
ZType

abstract property charge: int | float | Tensor

Charge of the system.

Returns
Charge of the system.

Return type
ZType

abstract property numel: int | float | Tensor

Total number of the electrons in the system.

Returns
Total number of the electrons in the system.

Return type
ZType

abstract property efield: Tuple[Tensor, ...] | None

External electric field of the system, or None if there is no electric field.

class RadialGrid(*ngrid: int*, *grid_integrator: str = 'chebyshev'*, *grid_transform: str | BaseGridTransform = 'logm3'*, *dtype: dtype = torch.float64*, *device: device = device(type='cpu')*)

Grid for radially symmetric system. This grid consists grid_integrator and grid_transform specifiers.

grid_integrator is to specify how to perform an integration on a fixed interval from -1 to 1.

grid_transform is to transform the integration from the coordinate of grid_integrator to the actual coordinate.

Examples

```
>>> grid = RadialGrid(100, grid_integrator="chebyshev",
...                   grid_transform="logm3")
>>> grid.get_rgrid().shape
torch.Size([100, 1])
>>> grid.get_dvolume().shape
torch.Size([100])
```

__init__(*ngrid: int*, *grid_integrator: str = 'chebyshev'*, *grid_transform: str | BaseGridTransform = 'logm3'*, *dtype: dtype = torch.float64*, *device: device = device(type='cpu')*)

Initialize the RadialGrid.

Parameters

- **ngrid** (*int*) – Number of grid points.
- **grid_integrator** (*str (default "chebyshev")*) – The grid integrator to use. Available options are "chebyshev", "chebyshev2", and "uniform".
- **grid_transform** (*Union[str, BaseGridTransform] (default "logm3")*) – The grid transformation to use. Available options are "logm3", "de2", and "treutlerm4".
- **dtype** (*torch.dtype, optional (default torch.float64)*) – The data type to use for the grid.
- **device** (*torch.device, optional (default torch.device('cpu'))*) – The device to use for the grid.

property coord_type

Returns the coordinate type of the grid.

Returns
The coordinate type of the grid. For RadialGrid, this is "radial".

Return type
str

property dtype

Returns the data type of the grid.

Returns
The data type of the grid.

Return type
torch.dtype

property device

Returns the device of the grid.

Returns
The device of the grid.

Return type
torch.device

get_dvolume() → Tensor

Returns the integration element of the grid.

Returns
The integration element of the grid.

Return type
torch.Tensor

get_rgrid() → Tensor

Returns the grid points.

Returns
The grid points.

Return type
torch.Tensor

__getitem__(*key: int | slice*) → *RadialGrid*

Returns a sliced RadialGrid.

Parameters
key (*Union[int, slice]*) – The index or slice to use for slicing the grid.

Returns
The sliced RadialGrid.

Return type
RadialGrid

getparamnames(*methodname: str*, *prefix: str = ''*)

Returns the parameter names for the given method.

Parameters

- **methodname** (*str*) – The name of the method.
- **prefix** (*str, optional (default "")*) – The prefix to use for the parameter names.

Returns
The parameter names for the given method.

Return type
List[str]

class **get_xw_integration**(*n: int*, *s0: str*)

returns `n` points of integration from -1 to 1 and its integration weights

Examples

```
>>> x, w = get_xw_integration(100, "chebyshev")
>>> x.shape
(100,)
>>> w.shape
(100,)
```

Parameters

- **n** (*int*) – Number of grid points.
- **s0** (*str*) – The grid integrator to use. Available options are *chebyshev*, *chebyshev2*, and *uniform*.

Returns
The integration points and weights.

Return type
Tuple[np.ndarray, np.ndarray]

References

class SlicedRadialGrid(*obj: RadialGrid*, *key: slice*)

Internal class to represent the sliced radial grid

__init__(*obj: RadialGrid*, *key: slice*)

Initialize the SlicedRadialGrid.

Parameters

- **obj** (RadialGrid) – The original RadialGrid.
- **key** (*slice*) – The slice to use for slicing the grid.

class BaseGridTransform

Base class for grid transformation Grid transformation is to transform the integration from the coordinate of grid_integrator to the actual coordinate.

It is used as a base class for other grid transformations. x2r and get_drdx are abstract methods that need to be implemented.

abstract x2r(*x: Tensor*) → Tensor

Transform from x to r coordinate

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
r – The coordinate from 0 to inf.

Return type
torch.Tensor

abstract get_drdx(*x: Tensor*) → Tensor

Returns the dr/dx

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
drdx – The dr/dx.

Return type
torch.Tensor

class DE2Transformation(*alpha: float = 1.0*, *rmin: float = 1e-07*, *rmax: float = 20*)

Double exponential formula grid transformation

Examples

```
>>> x = torch.linspace(-1, 1, 100)
>>> r = DE2Transformation().x2r(x)
>>> r.shape
torch.Size([100])
>>> drdx = DE2Transformation().get_drdx(x)
>>> drdx.shape
torch.Size([100])
```

References

__init__(*alpha: float = 1.0*, *rmin: float = 1e-07*, *rmax: float = 20*)

x2r(*x: Tensor*) → Tensor

Transform from x to r coordinate

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
r – The coordinate from 0 to inf.

Return type
torch.Tensor

get_drdx(*x: Tensor*) → Tensor

Returns the dr/dx

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
drdx – The dr/dx.

Return type
torch.Tensor

class LogM3Transformation(*ra: float = 1.0*, *eps: float = 1e-15*)

LogM3 grid transformation

Examples

```
>>> x = torch.linspace(-1, 1, 100)
>>> r = LogM3Transformation().x2r(x)
>>> r.shape
torch.Size([100])
>>> drdx = LogM3Transformation().get_drdx(x)
>>> drdx.shape
torch.Size([100])
```

References

`__init__`(*ra: float = 1.0*, *eps: float = 1e-15*)

Initialize the LogM3Transformation.

Parameters

- **ra** (*float (default 1.0)*) – The parameter to control the range of the grid.
- **eps** (*float (default 1e-15)*) – The parameter to avoid numerical instability.

`x2r`(*x: Tensor*) → Tensor

Transform from x to r coordinate

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
The coordinate from 0 to inf.

Return type
torch.Tensor

`get_drdx`(*x: Tensor*) → Tensor

Returns the dr/dx

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
The dr/dx.

Return type
torch.Tensor

class `TreutlerM4Transformation`(*xi: float = 1.0*, *alpha: float = 0.6*, *eps: float = 1e-15*)

Treutler M4 grid transformation

Examples

```
>>> x = torch.linspace(-1, 1, 100)
>>> r = TreutlerM4Transformation().x2r(x)
>>> r.shape
torch.Size([100])
>>> drdx = TreutlerM4Transformation().get_drdx(x)
>>> drdx.shape
torch.Size([100])
```

References

`__init__`(*xi: float = 1.0*, *alpha: float = 0.6*, *eps: float = 1e-15*)

Initialize the TreutlerM4Transformation.

Parameters

- **xi** (*float (default 1.0)*) – The parameter to control the range of the grid.
- **alpha** (*float (default 0.6)*) – The parameter to control the range of the grid.
- **eps** (*float (default 1e-15)*) – The parameter to avoid numerical instability.

`x2r`(*x: Tensor*) → Tensor

Transform from x to r coordinate

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
The coordinate from 0 to inf.

Return type
torch.Tensor

`get_drdx`(*x: Tensor*) → Tensor

Returns the dr/dx

Parameters
x (*torch.Tensor*) – The coordinate from -1 to 1.

Returns
The dr/dx.

Return type
torch.Tensor

class `get_grid_transform`(*s0: str* | BaseGridTransform)

grid transformation object from the input

Examples

```
>>> transform = get_grid_transform("logm3")
>>> transform.x2r(torch.tensor([0.5]))
tensor([2.])
```

Parameters
s0 (*Union[str, BaseGridTransform]*) – The grid transformation to use. Available options are *logm3*, *de2*, and *treutlerm4*.

Returns
The grid transformation object.

Return type
BaseGridTransform

Raises
RuntimeError – If the input is not a valid grid transformation.

class HF(*system: BaseSystem*, *restricted: bool | None = None*, *variational: bool = False*)
Performing Restricted or Unrestricted Kohn-Sham DFT calculation.

Parameters

- **system** (*BaseSystem*) – The system to be calculated.
- **restricted** (*bool or None*) – If True, performing restricted Kohn-Sham DFT. If False, it performs the unrestricted Kohn-Sham DFT. If None, it will choose True if the system is unpolarized and False if it is polarized
- **variational** (*bool*) – If True, then use optimization of the free orbital parameters to find the minimum energy. Otherwise, use self-consistent iterations.

__init__(*system: BaseSystem*, *restricted: bool | None = None*, *variational: bool = False*)
Initialize the SCF calculation.

Parameters

- **engine** (*BaseSCFEngine*) – SCF engine
- **variational** (*bool*) – If True, then use optimization of the free orbital parameters to find the minimum energy. Otherwise, use self-consistent iterations.

class HFEngine(*system: BaseSystem*, *restricted: bool | None = None*, *build_grid_if_necessary: bool = False*)
Engine to be used with Hartree Fock. This class provides the calculation of the self-consistency iteration step and the calculation of the post-calculation properties.

Examples

```
>>> import torch
>>> from deepchem.utils.dft_utils import HFEngine, BaseSystem, BaseHamilton,␣
↪SpinParam, BaseGrid
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> from typing import List, Optional
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
```

(continues on next page)

(continued from previous page)

```
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
...     def _rmv(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _mm(self, x):
...         return torch.matmul(self.param, x)
...     def _rmm(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _fullmatrix(self):
...         return self.param
>>> class MyHamilton(BaseHamilton):
...     def __init__(self):
...         self._nao = 2
...         self._kpts = torch.tensor([[0.0, 0.0, 0.0]])
...         self._df = None
...     @property
...     def nao(self):
...         return self._nao
...     @property
...     def kpts(self):
...         return self._kpts
...     @property
...     def df(self):
...         return self._df
...     def build(self):
...         return self
...     def get_nuclattr(self):
...         return torch.ones((1, 1, self.nao, self.nao))
...     def get_e_elrep(self, dmtot):
...         return 2 * dmtot
...     def get_e_exchange(self, dm):
...         if isinstance(dm, SpinParam):
...             return SpinParam.sum(dm)
...         else:
...             return 2 * dm
...     def get_e_hcore(self, dm):
...         return 4 * dm
...     def get_elrep(self, dmtot):
...         return MyLinOp((self.nao + 1, self.nao + 1))
...     def get_exchange(self, dm):
...         return MyLinOp((self.nao + 1, self.nao + 1))
...     def get_kinnucl(self):
...         linop = MyLinOp((self.nao + 1, self.nao + 1))
...         return linop
...     def ao_orb2dm(self, orb: torch.Tensor,
...                   orb_weight: torch.Tensor) -> torch.Tensor:
...         return orb * orb_weight
...     def ao_orb_params2dm(
...         self,
```

(continues on next page)

(continued from previous page)

```
...         ao_orb_params: torch.Tensor,
...         ao_orb_coeffs: torch.Tensor,
...         orb_weight: torch.Tensor,
...         with_penalty: Optional[float] = None) -> List[torch.Tensor]:
...         return [ao_orb_params * orb_weight, ao_orb_coeffs * orb_weight]
>>> ham = MyHamilton()
>>> class MySystem(BaseSystem):
...     def __init__(self):
...         self.hamiltonian = ham
...         self.grid = BaseGrid()
...     def get_hamiltonian(self):
...         return self.hamiltonian
...     def get_grid(self):
...         return self.grid
...     def requires_grid(self):
...         return True
...     def get_orbweight(self, polarized: bool = False) -> Union[torch.Tensor,␣
↪SpinParam[torch.Tensor]]:
...         return SpinParam(torch.tensor([2.0]), torch.tensor([2.0]))
...     def get_nuclei_energy(self):
...         return torch.tensor(10.0)
>>> system = MySystem()
>>> engine = HFEngine(system, False)
>>> engine.set_eigen_options(eigen_options={"method": "exacteig"})
>>> engine.dm2energy(torch.tensor([2])).shape
torch.Size([1])
>>> engine.dm2scp(torch.tensor([2])).shape
torch.Size([3, 3])
>>> engine.scp2dm(torch.rand((2, 2, 2))).u.shape
torch.Size([2, 1])
```

`__init__`(*system: BaseSystem*, *restricted: bool | None = None*, *build_grid_if_necessary: bool = False*)

`get_system`() → *BaseSystem*

Return the system object.

Returns

The system object.

Return type

BaseSystem

`property shape`

Shape of the density matrix

Returns

Shape of the density matrix.

Return type

Tuple[int, int]

`property dtype`

Dtype of the density matrix

Returns

Dtype of the density matrix.

Return type
torch.dtype

property device

Device of the density matrix

Returns
Device of the density matrix.

Return type
torch.device

property polarized

Returns if the calculation is polarized

Returns
If the calculation is polarized.

Return type
bool

dm2scp(*dm: Tensor* | SpinParam*[Tensor]*) → Tensor

Convert from density matrix to a self-consistent parameter (scp)

Parameters
dm (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix to be converted.

Returns
Self-consistent parameter.

Return type
torch.Tensor

scp2dm(*scp: Tensor*) → Tensor | *SpinParam*[Tensor]

Convert from self-consistent parameter (scp) to density matrix

Parameters
scp (*torch.Tensor*) – Self-consistent parameter to be converted.

Returns
Density matrix.

Return type
Union[torch.Tensor, *SpinParam*[torch.Tensor]]

scp2scp(*scp: Tensor*) → Tensor

Self-consistent iteration step from a self-consistent parameter (scp) to an scp

Parameters
scp (*torch.Tensor*) – Self-consistent parameter to be converted.

Returns
New self-consistent parameter.

Return type
torch.Tensor

aoparams2ene(*aoparams: Tensor*, *aocoeffs: Tensor*, *with_penalty: float* | *None = None*) → Tensor

Calculate the energy from the atomic orbital params

Parameters

- **aoparams** (*torch.Tensor*) – Atomic orbital parameters.
- **aocoeffs** (*torch.Tensor*) – Atomic orbital coefficients.
- **with_penalty** (*Optional[float]*) – Penalty factor to be added to the energy.

Returns
Energy value.

Return type
torch.Tensor

aoparams2dm(*aoparams: Tensor*, *aocoeffs: Tensor*, *with_penalty: float | None = None*) → Tuple[Tensor | *SpinParam*[Tensor], Tensor | None]

Convert the aoparams to density matrix and penalty factor

Parameters

- **aoparams** (*torch.Tensor*) – Atomic orbital parameters.
- **aocoeffs** (*torch.Tensor*) – Atomic orbital coefficients.
- **with_penalty** (*Optional[float]*) – Penalty factor to be added to the energy.

Returns
Density matrix and the penalty factor.

Return type
Tuple[Union[torch.Tensor, *SpinParam*[torch.Tensor]], Optional[torch.Tensor]]

pack_aoparams(*aoparams: Tensor* | SpinParam*[Tensor]*) → Tensor

Check if polarized, then pack it by concatenating them in the last dimension

Parameters
aoparams (*Union[torch.Tensor,* SpinParam*[torch.Tensor]]*) – Atomic orbital parameters.

Returns
Packed atomic orbital parameters.

Return type
torch.Tensor

unpack_aoparams(*aoparams: Tensor*) → Tensor | *SpinParam*[Tensor]

Check if polarized, then construct the SpinParam (reverting the pack_aoparams)

Parameters
aoparams (*torch.Tensor*) – Packed atomic orbital parameters.

Returns
Atomic orbital parameters.

Return type
Union[torch.Tensor, *SpinParam*[torch.Tensor]]

set_eigen_options(*eigen_options: Dict[str, Any]*) → None

Set the eigendecomposition (diagonalization) option

Parameters
eigen_options (*Dict[str, Any]*) – Options for the eigendecomposition.

dm2energy(*dm: Tensor* | SpinParam*[Tensor]*) → Tensor

Calculate the energy given the density matrix

Parameters
: **dm** (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Density matrix.

Returns
: Energy.

Return type
: torch.Tensor

diagonalize(*fock:* LinearOperator | SpinParam*[*LinearOperator*]*, *norb: int* | SpinParam*[int]*) → Tuple[Tensor, Tensor] | Tuple[*SpinParam*[Tensor], *SpinParam*[Tensor]]

Diagonalize the fock matrix

Parameters

- **fock** (*Union[LinearOperator, SpinParam[LinearOperator]]*) – Fock matrix.
- **norb** (*Union[int, SpinParam[int]]*) – Number of orbitals.

Returns
: Eigenvalues and eigenvectors.

Return type
: Union[Tuple[torch.Tensor, torch.Tensor], Tuple[*SpinParam*[torch.Tensor], *SpinParam*[torch.Tensor]]]

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

Parameter names for the given method

Parameters

- **methodname** (*str*) – Method name.
- **prefix** (*str*) – Prefix to be added to the parameter names.

Returns
: Parameter names.

Return type
: List[str]

class BaseQCCalc

Quantum Chemistry calculation. This class is the interface to the users regarding parameters that can be calculated after the self-consistent iterations (or other processes).

This object is usually a thin-wrapper of an engine class where the self-consistent iteration steps (and other processes) are described. To avoid known memory leak, the self-consistent iteration should be run from this object while the steps should be described in another object (i.e. the engine). Details about the leak: https://github.com/pytorch/pytorch/issues/52140

Examples

```
>>> from deepchem.utils.dft_utils import BaseSystem, BaseQCCalc
>>> class DummyQCCalc(BaseQCCalc):
...     def __init__(self, system: BaseSystem):
...         self.system = system
...     def get_system(self) -> BaseSystem:
...         return self.system
...     def run(self, **kwargs):
...         pass
...     def energy(self) -> torch.Tensor:
...         return torch.tensor(0.0)
>>> system = BaseSystem()
>>> qc = DummyQCCalc(system)
>>> qc.energy()
tensor(0.)
```

abstract get_system() → *BaseSystem*

Returns the system in the QC calculation

Returns
The system that is being calculated.

Return type
BaseSystem

abstract run(***kwargs*)

Run the calculation.

Note that this method can be invoked several times for one object to try for various self-consistent options to reach convergence.

abstract energy() → Tensor

Obtain the energy of the system.

Returns
The energy of the system.

Return type
torch.Tensor

abstract aodm() → Tensor | *SpinParam*[Tensor]

Returns the density matrix in atomic orbital. For polarized case, it returns a SpinParam of 2 tensors representing the density matrices for spin-up and spin-down.

Returns
The density matrix in atomic orbital. Shape: (nao, nao)

Return type
Union[torch.Tensor, *SpinParam*[torch.Tensor]]

abstract dm2energy(*dm: Tensor* | SpinParam*[Tensor]*) → Tensor

Calculate the energy from the given density matrix.

Parameters
dm (*Union[torch.Tensor,* SpinParam*[torch.Tensor]]*) – The input density matrix. It is tensor if restricted, and SpinParam of tensor if unrestricted.

Returns
Tensor that represents the energy given the energy.

Return type
torch.Tensor

abstract getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

Return a list with the parameter names corresponding to the given method (methodname)

Parameters
methodname (*str*) – The name of the method to get the parameter names from.

Returns
List of parameter names of methodname

Return type
List[str]

class SCF_QCCalc(*engine: BaseSCFEngine*, *variational: bool = False*)

Performing Restricted or Unrestricted self-consistent field iteration (e.g. Hartree-Fock or Density Functional Theory)

Examples

```
>>> from deepchem.utils.dft_utils import SCF_QCCalc, BaseSCFEngine
>>> from deepchem.utils.dft_utils import SpinParam
```

# Define the engine >>> class engine(BaseSCFEngine): … def polarised(): … return False … def dm2energy(self, dm: torch.Tensor | SpinParam[torch.Tensor]) -> torch.Tensor: … return dm * 1.1 >>> myEngine = engine() >>> a = SCF_QCCalc(myEngine) >>> a.dm2energy(torch.tensor([1.1])) tensor([1.2100])

__init__(*engine: BaseSCFEngine*, *variational: bool = False*)

Initialize the SCF calculation.

Parameters

- **engine** (*BaseSCFEngine*) – SCF engine
- **variational** (*bool*) – If True, then use optimization of the free orbital parameters to find the minimum energy. Otherwise, use self-consistent iterations.

get_system() → *BaseSystem*

Returns the system in the QC calculation

Returns
System that is being calculated.

Return type
BaseSystem

run(*dm0: str | Tensor | SpinParam[Tensor] | None = '1e'*, *eigen_options: Dict[str, Any] | None = None*, *fwd_options: Dict[str, Any] | None = None*, *bck_options: Dict[str, Any] | None = None*) → *BaseQCCalc*

Run the calculation.

Note that this method can be invoked several times for one object to try for various self-consistent options to reach convergence.

Parameters

- **dm0** (*Optional[Union[str, torch.Tensor, SpinParam[torch.Tensor]]]*) – Initial density matrix. If it is a string, it can be "1e" to use the 1-electron Hamiltonian to generate the initial density matrix.
- **eigen_options** (*Optional[Dict[str, Any]]*) – Options for the diagonalization (i.e. eigendecomposition).
- **fwd_options** (*Optional[Dict[str, Any]]*) – Options for the forward iteration (i.e. the iteration to find the minimum energy).
- **bck_options** (*Optional[Dict[str, Any]]*) – Options for the backward iteration (i.e. the iteration to find the minimum energy).

`energy()` → Tensor

Obtain the energy of the system.

Returns
Energy of the system.

Return type
torch.Tensor

`aodm()` → Tensor | *SpinParam*[Tensor]

Returns the density matrix in atomic orbital. For polarized case, it returns a SpinParam of 2 tensors representing the density matrices for spin-up and spin-down.

Returns
Density matrix in atomic orbital. Shape: (nao, nao)

Return type
Union[torch.Tensor, *SpinParam*[torch.Tensor]]

`dm2energy`(*dm: Tensor* | SpinParam*[Tensor]*)

Calculate the energy from the given density matrix.

Parameters
dm (*Union[torch.Tensor, SpinParam[torch.Tensor]]*) – Input density matrix. It is tensor if restricted, and SpinParam of tensor if unrestricted.

Returns
Tensor that represents the energy given the energy.

Return type
torch.Tensor

`class EditableModule`

EditableModule is a base class to enable classes that it inherits be converted to pure functions for higher order derivatives purpose.

Usage

To use this class, the user must implement the `getparamnames` method which returns a list of tensor names that affect the output of the method with name indicated in `methodname`.

Used in:

- Classes of Density Functional Theory (DFT).
- It can also be used in other classes that need to be converted to pure functions for higher order derivatives purpose.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import EditableModule
>>> class A(EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
>>> a = torch.tensor(2.0).requires_grad_()
>>> x = torch.tensor(0.4).requires_grad_()
>>> alpha = A(a)
>>> alpha.mult(x)
tensor(1.6000, grad_fn=<MulBackward0>)
>>> alpha.getparamnames("mult")
['b']
>>> alpha.assertparams(alpha.mult, x)
"mult" method check done
```

getparams(*methodname: str*) → Sequence[Tensor]

Returns a list of tensor parameters used in the object's operations. Requires the `getparamnames` method to be implemented.

Parameters
methodname (*str*) – The name of the method of the class.

Returns
Sequence of tensors that are involved in the specified method of the object.

Return type
Sequence[torch.Tensor]

setparams(*methodname: str*, **params*) → int

Set the input parameters to the object's parameters to make a copy of the operations.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- ***params** – The parameters to be set to the object's parameters.

Returns
The number of parameters that are set to the object's parameters.

Return type
int

cached_getparamnames(*methodname: str*, *refresh: bool = False*) → List[str]

getparamnames, but cached, so it is only called once

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **refresh** (*bool*) – If True, the cache is refreshed.

Returns
Sequence of name of parameters affecting the output of the method.

Return type
List[str]

abstract **getparamnames**(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns
Sequence of name of parameters affecting the output of the method.

Return type
List[str]

Raises
KeyError – If the list in this function does not contain `methodname`.

getuniqueparams(*methodname: str*, *onlyleaves: bool = False*) → List[Tensor]

Returns the list of unique parameters involved in the method specified by *methodname*.

Parameters

- **methodname** (*str*) – Name of the method where the returned parameters play roles.
- **onlyleaves** (*bool*) – If True, only returns leaf tensors. Otherwise, returns all tensors.

Returns
List of tensors that are involved in the specified method of the object.

Return type
List[torch.Tensor]

setuniqueparams(*methodname: str*, **uniqueparams*) → int

Set the input parameters to the object's parameters to make a copy of the operations. The input parameters are unique parameters, i.e. they are not necessarily the same tensors as the object's parameters.

Note: This function can only be run after running getuniqueparams.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- ***uniqueparams** – The parameters to be set to the object's parameters. The number of parameters must be the same as the number of unique parameters returned by `getuniqueparams`.

Returns
The number of parameters that are set to the object's parameters.

Return type
int

assertparams(*method: Callable*, **args*, ***kwargs*)

Perform a rigorous check on the implemented `getparamnames` in the class for a given method and its parameters as well as keyword Parameters. It raises warnings if there are missing or excess parameters in the `getparamnames` implementation.

Parameters

- **method** (*Callable*) – The method of this class to be tested
- ***args** – Parameters of the method
- ****kwargs** – Keyword parameters of the method

normalize_bcast_dims(**shapes*)

Normalize the lengths of the input shapes to have the same length. The shapes are padded at the front by 1 to make the lengths equal.

Examples

```
>>> normalize_bcast_dims([1, 2, 3], [2, 3])
[[1, 2, 3], [1, 2, 3]]
```

Parameters
shapes (*List[List[int]]*) – The shapes to normalize.

Returns
The normalized shapes.

Return type
List[List[int]]

get_bcasted_dims(**shapes*)

Return the broadcasted shape of the given shapes.

Examples

```
>>> get_bcasted_dims([1, 2, 5], [2, 3, 4])
[2, 3, 5]
```

Parameters
shapes (*List[List[int]]*) – The shapes to broadcast.

Returns
The broadcasted shape.

Return type
List[int]

match_dim(**xs: Tensor*, *contiguous: bool = False*) → Tuple[Tensor, ...]

match the N-1 dimensions of x and xq for searchsorted and gather with dim=-1

Examples

```
>>> x = torch.randn(10, 5)
>>> xq = torch.randn(10, 3)
>>> x_new, xq_new = match_dim(x, xq)
>>> x_new.shape
torch.Size([10, 5])
>>> xq_new.shape
torch.Size([10, 3])
```

class LinearOperator(**args*, ***kwargs*)

`LinearOperator` is a base class designed to behave as a linear operator without explicitly determining the matrix. This `LinearOperator` should be able to operate as batched linear operators where its shape is `(B1,B2,...,Bb,p,q)` with `B*` as the (optional) batch dimensions. For a user-defined class to behave as `LinearOperator`, it must use `LinearOperator` as one of the parent and it has to have `._mv()` method implemented and `._getparamnames()` if used in xitorch's functionals with torch grad enabled.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
...     def _rmv(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _mm(self, x):
...         return torch.matmul(self.param, x)
...     def _rmm(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _fullmatrix(self):
...         return self.param
>>> linop = MyLinOp((1,3,1,2))
>>> print(linop)
LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32, device =␣
↪cpu
>>> x = torch.rand(1,3,2,2)
>>> linop.mv(x)
tensor([[[[0.1991, 0.1011]],

         [[0.3764, 0.5742]],

         [[1.0345, 1.1802]]]])
>>> x = torch.rand(1,3,1,1)
>>> linop.rmv(x)
tensor([[[[0.0250],
          [0.1827]],
```

(continues on next page)

(continued from previous page)

```
         [[0.0794],
          [0.1463]],

         [[0.1207],
          [0.1345]]]])
>>> x = torch.rand(1,3,2,2)
>>> linop.mm(x)
tensor([[[[0.8891, 0.4243]],

         [[0.4856, 0.3128]],

         [[0.6601, 0.9532]]]])
>>> x = torch.rand(1,3,1,2)
>>> linop.rmm(x)
tensor([[[[0.0473, 0.0019],
          [0.3455, 0.0138]],

         [[0.0580, 0.2504],
          [0.1069, 0.4614]],

         [[0.4779, 0.1102],
          [0.5326, 0.1228]]]])
>>> linop.fullmatrix()
tensor([[[[0.1117, 0.8158]],

         [[0.2626, 0.4839]],

         [[0.6765, 0.7539]]]])
```

static __new__(*self*, **args*, ***kwargs*)

Check the implemented functions in the class.

classmethod m(*mat: Tensor*, *is_hermitian: bool | None = None*)

Class method to wrap a matrix into `LinearOperator`.

Parameters

- **mat** (*torch.Tensor*) – Matrix to be wrapped in the `LinearOperator`.
- **is_hermitian** (*bool or None*) – Indicating if the matrix is Hermitian. If `None`, the symmetry will be checked. If supplied as a bool, there is no check performed.

Returns

Linear operator object that represents the matrix.

Return type

LinearOperator

Example

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> seed = torch.manual_seed(100)
>>> mat = torch.rand(1,3,1,2)  # 1x2 matrix with (1,3) batch dimensions
>>> linop = LinearOperator.m(mat)
>>> print(linop)
MatrixLinearOperator with shape (1, 3, 1, 2):
   tensor([[[[0.1117, 0.8158]],

            [[0.2626, 0.4839]],

            [[0.6765, 0.7539]]]])
```

`__init__`(*shape: Sequence[int]*, *is_hermitian: bool = False*, *dtype: dtype | None = None*, *device: device | None = None*, *_suppress_hermit_warning: bool = False*) → None

Initialize the `LinearOperator`.

Parameters

- **shape** (*Sequence[int]*) – The shape of the linear operator.
- **is_hermitian** (*bool*) – Whether the linear operator is Hermitian.
- **dtype** (*torch.dtype or None*) – The dtype of the linear operator.
- **device** (*torch.device or None*) – The device of the linear operator.
- **_suppress_hermit_warning** (*bool*) – Whether to suppress the warning when the linear operator is Hermitian but the `.rmv()` or `.rmm()` is implemented.

`getlinopparams`() → Sequence[Tensor]

Get the parameters that affects most of the methods (i.e. mm, mv, rmm, rmv).

`uselinopparams`(**params*)

Context manager to temporarily set the parameters that affects most of the methods (i.e. mm, mv, rmm, rmv).

`mv`(*x: Tensor*) → Tensor

Apply the matrix-vector operation to vector `x` with shape `(...,q)`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters

x (*torch.tensor*) – The vector with shape `(...,q)` where the linear operation is operated on

Returns

y – The result of the linear operation with shape `(...,p)`

Return type

torch.tensor

`mm`(*x: Tensor*) → Tensor

Apply the matrix-matrix operation to matrix `x` with shape `(...,q,r)`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters

x (*torch.tensor*) – The matrix with shape `(...,q,r)` where the linear operation is operated on.

Returns
y – The result of the linear operation with shape `(...,p,r)`

Return type
torch.tensor

rmv(*x: Tensor*) → Tensor

Apply the matrix-vector adjoint operation to vector `x` with shape `(...,p)`, i.e. `A^H x`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters
x (*torch.tensor*) – The vector of shape `(...,p)` where the adjoint linear operation is operated at.

Returns
y – The result of the adjoint linear operation with shape `(...,q)`

Return type
torch.tensor

rmm(*x: Tensor*) → Tensor

Apply the matrix-matrix adjoint operation to matrix `x` with shape `(...,p,r)`, i.e. `A^H X`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters
x (*torch.Tensor*) – The matrix of shape `(...,p,r)` where the adjoint linear operation is operated on.

Returns
y – The result of the adjoint linear operation with shape `(...,q,r)`.

Return type
torch.Tensor

fullmatrix() → Tensor

Full matrix representation of the linear operator.

scipy_linalg_op()

Return the scipy.sparse.linalg.LinearOperator object of the linear operator.

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

Get the parameter names that affects the method.

property H

Returns a LinearOperator representing the Hermite / transposed of the self LinearOperator.

Returns
The Hermite / transposed LinearOperator

Return type
LinearOperator

matmul(*b:* LinearOperator, *is_hermitian: bool = False*)

Returns a LinearOperator representing *self @ b*.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
>>> linop1 = MyLinOp((1,3,1,2))
>>> linop2 = MyLinOp((1,3,2,1))
>>> linop = linop1.matmul(linop2)
>>> print(linop)
MatmulLinearOperator with shape (1, 3, 1, 1) of:
 * LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32,␣
↪device = cpu
 * LinearOperator (MyLinOp) with shape (1, 3, 2, 1), dtype = torch.float32,␣
↪device = cpu
>>> x = torch.rand(1,3,1,1)
>>> linop.mv(x)
tensor([[[[0.0458]],

         [[0.0880]],

         [[0.2664]]]])
```

Parameters

- **b** (LinearOperator) – Other linear operator
- **is_hermitian** (*bool*) – Flag to indicate if the resulting LinearOperator is Hermitian.

Returns
LinearOperator representing *self @ b*

Return type
LinearOperator

__add__(*b:* LinearOperator)
Addition with another linear operator.

Examples

```
>>> class Operator(LinearOperator):
...     def __init__(self, mat: torch.Tensor, is_hermitian: bool) -> None:
...         super(Operator, self).__init__(
...             shape=mat.shape,
...             is_hermitian=is_hermitian,
...             dtype=mat.dtype,
...             device=mat.device,
```

(continues on next page)

(continued from previous page)

```
...             _suppress_hermit_warning=True,
...         )
...         self.mat = mat
...     def _mv(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat, x.unsqueeze(-1)).squeeze(-1)
...     def _mm(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat, x)
...     def _rmv(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat.transpose(-3, -1).conj(), x.unsqueeze(-
↪1)).squeeze(-1)
...     def _rmm(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat.transpose(-2, -1).conj(), x)
...     def _fullmatrix(self) -> torch.Tensor:
...         return self.mat
...     def _getparamnames(self, prefix: str = "") -> List[str]:
...         return [prefix + "mat"]
>>> op = Operator(torch.tensor([[1, 2.],
...                             [3, 4]]), is_hermitian=False)
>>> x = torch.tensor([[2, 2],
...                   [1, 2.]])
>>> op.mm(x)
tensor([[ 4.,  6.],
        [10., 14.]])
>>> op2 = op + op
>>> op2.mm(x)
tensor([[ 8., 12.],
        [20., 28.]])
```

Parameters
: **b** (LinearOperator) – The linear operator to be added.

Returns
: The result of the addition.

Return type
: *LinearOperator*

__sub__(*b:* LinearOperator)

Subtraction with another linear operator.

Examples

```
>>> class Operator(LinearOperator):
...     def __init__(self, mat: torch.Tensor, is_hermitian: bool) -> None:
...         super(Operator, self).__init__(
...             shape=mat.shape,
...             is_hermitian=is_hermitian,
...             dtype=mat.dtype,
...             device=mat.device,
...             _suppress_hermit_warning=True,
...         )
```

(continues on next page)

(continued from previous page)

```
...         self.mat = mat
...     def _mv(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat, x.unsqueeze(-1)).squeeze(-1)
...     def _mm(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat, x)
...     def _rmv(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat.transpose(-3, -1).conj(), x.unsqueeze(-
↪1)).squeeze(-1)
...     def _rmm(self, x: torch.Tensor) -> torch.Tensor:
...         return torch.matmul(self.mat.transpose(-2, -1).conj(), x)
...     def _fullmatrix(self) -> torch.Tensor:
...         return self.mat
...     def _getparamnames(self, prefix: str = "") -> List[str]:
...         return [prefix + "mat"]
>>> op = Operator(torch.tensor([[1, 2.],
...                             [3, 4]]), is_hermitian=False)
>>> op1 = Operator(torch.tensor([[0, 1.],
...                              [1, 2]]), is_hermitian=False)
>>> x = torch.tensor([[2, 2],
...                   [1, 2.]])
>>> op.mm(x)
tensor([[ 4.,  6.],
        [10., 14.]])
>>> op2 = op - op1
>>> op2.mm(x)
tensor([[3., 4.],
        [6., 8.]])
```

Parameters
: **b** (`LinearOperator`) – The linear operator to be subtracted.

Returns
: The result of the subtraction.

Return type
: *LinearOperator*

property dtype: dtype
: The dtype of the linear operator.

property device: device
: The device of the linear operator.

property shape: Sequence[int]
: The shape of the linear operator.

property is_hermitian: bool
: Whether the linear operator is Hermitian.

property is_mv_implemented: bool
: Whether the `.mv()` method is implemented.

property is_mm_implemented: bool
: Whether the `.mm()` method is implemented.

property is_rmv_implemented: bool

Whether the `.rmv()` method is implemented.

property is_rmm_implemented: bool

Whether the `.rmm()` method is implemented.

property is_fullmatrix_implemented: bool

Whether the `.fullmatrix()` method is implemented.

property is_getparamnames_implemented: bool

Whether the `._getparamnames()` method is implemented.

class AddLinearOperator(**args*, ***kwargs*)

Adds two linear operators.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
...     def _rmv(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _mm(self, x):
...         return torch.matmul(self.param, x)
...     def _rmm(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
...     def _fullmatrix(self):
...         return self.param
>>> linop1 = MyLinOp((1,3,1,2))
>>> linop2 = MyLinOp((1,3,1,2))
>>> linop = AddLinearOperator(linop1, linop2)
>>> print(linop)
AddLinearOperator with shape (1, 3, 1, 2) of:
 * LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32, device␣
↪= cpu
 * LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32, device␣
↪= cpu
>>> x = torch.rand(1,3,2,2)
>>> linop.mv(x)
tensor([[[[0.6256, 1.0689]],

         [[0.6039, 0.5380]],

         [[0.9702, 2.1129]]]])
>>> x = torch.rand(1,3,1,1)
>>> linop.rmv(x)
```

(continues on next page)

(continued from previous page)

```
tensor([[[[0.1662],
          [0.3813]],

         [[0.4460],
          [0.5705]],

         [[0.5942],
          [1.1089]]]])
>>> x = torch.rand(1,2,2,1)
>>> linop.mm(x)
tensor([[[[0.7845],
          [0.5439]],


        [[[0.6518],
          [0.4318]],


        [[[1.4336],
          [0.9796]]]])
```

`__init__`(*a:* LinearOperator, *b:* LinearOperator, *mul: int = 1*)

Initialize the `AddLinearOperator`.

Parameters

- **a** (`LinearOperator`) – The first linear operator to be added.
- **b** (`LinearOperator`) – The second linear operator to be added.
- **mul** (*int*) – The multiplier of the second linear operator. Default to 1. If -1, then the second linear operator will be subtracted.

class MulLinearOperator(**args*, ***kwargs*)

Multiply a linear operator with a scalar.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
>>> linop = MyLinOp((1,3,1,2))
>>> print(linop)
LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32, device =␣
↪cpu
>>> x = torch.rand(1,3,2,2)
```

(continues on next page)

(continued from previous page)

```
>>> linop.mv(x)
tensor([[[[0.1991, 0.1011]],

         [[0.3764, 0.5742]],

         [[1.0345, 1.1802]]]])
>>> linop2 = linop * 2
>>> linop2.mv(x)
tensor([[[[0.3981, 0.2022]],

         [[0.7527, 1.1485]],

         [[2.0691, 2.3603]]]])
```

`__init__`(*a:* LinearOperator, *f: int | float*)

Initialize the MulLinearOperator.

Parameters

- **a** (`LinearOperator`) – Linear operator to be multiplied.
- **f** (*`Union[int, float]`*) – Integer or floating point number to be multiplied.

class AdjointLinearOperator(**args*, ***kwargs*)

Adjoint of a LinearOperator.

This is used to calculate the adjoint of a LinearOperator without explicitly constructing the adjoint matrix. This is useful when the adjoint matrix is not explicitly constructed, e.g. when the LinearOperator is a function of other parameters.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
...     def _rmv(self, x):
...         return torch.matmul(self.param.transpose(-2,-1).conj(), x)
>>> linop = MyLinOp((1,3,1,2))
>>> print(linop)
LinearOperator (MyLinOp) with shape (1, 3, 1, 2), dtype = torch.float32, device =␣
↪cpu
>>> x = torch.rand(1,3,1,1)
>>> linop.rmv(x)
tensor([[[[0.0293],
          [0.2143]],
```

(continues on next page)

(continued from previous page)

```
          [[0.0112],
           [0.0207]],

          [[0.1407],
           [0.1568]]]])
>>> linop2 = linop.H
>>> linop2.mv(x)
tensor([[[[0.0293],
           [0.2143]],

          [[0.0112],
           [0.0207]],

          [[0.1407],
           [0.1568]]]])
```

`__init__`(*obj: LinearOperator*)

Initialize the `AdjointLinearOperator`.

Parameters
obj (`LinearOperator`) – The linear operator to be adjointed.

property H

Adjoint of the linear operator.

Returns
Adjoint of the linear operator.

Return type
LinearOperator

class MatmulLinearOperator(**args*, ***kwargs*)

Matrix-matrix multiplication of two linear operators.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> class MyLinOp(LinearOperator):
...     def __init__(self, shape):
...         super(MyLinOp, self).__init__(shape)
...         self.param = torch.rand(shape)
...     def _getparamnames(self, prefix=""):
...         return [prefix + "param"]
...     def _mv(self, x):
...         return torch.matmul(self.param, x)
>>> linop1 = MyLinOp((1,3,2,2))
>>> linop2 = MyLinOp((1,3,2,2))
>>> linop = MatmulLinearOperator(linop1, linop2)
>>> print(linop)
MatmulLinearOperator with shape (1, 3, 2, 2) of:
 * LinearOperator (MyLinOp) with shape (1, 3, 2, 2), dtype = torch.float32, device␣
↪= cpu
```

(continues on next page)

(continued from previous page)

```
 * LinearOperator (MyLinOp) with shape (1, 3, 2, 2), dtype = torch.float32, device␣
↪= cpu
>>> x = torch.rand(1,2,2,1)
>>> linop.mm(x)
tensor([[[[0.7998],
          [0.8016]],

         [[0.6515],
          [0.6835]]],


        [[[0.9251],
          [1.1611]],

         [[0.2781],
          [0.3609]]],


        [[[0.2591],
          [0.2376]],

         [[0.8009],
          [0.8087]]]]])
```

__init__(*a: LinearOperator*, *b: LinearOperator*, *is_hermitian: bool = False*)

Initialize the `MatmulLinearOperator`.

Parameters

- **a** (`LinearOperator`) – The first linear operator to be multiplied.
- **b** (`LinearOperator`) – The second linear operator to be multiplied.
- **is_hermitian** (*bool*) – Whether the result is Hermitian. Default to False.

class MatrixLinearOperator(**args*, ***kwargs*)

Class method to wrap a matrix into `LinearOperator`. It is a standard linear operator, used in many operations.

Examples

```
>>> import torch
>>> seed = torch.manual_seed(100)
>>> mat = torch.rand(3, 2)
>>> linop = MatrixLinearOperator(mat, is_hermitian=False)
>>> print(linop)
MatrixLinearOperator with shape (3, 2):
   tensor([[0.1117, 0.8158],
           [0.2626, 0.4839],
           [0.6765, 0.7539]])
>>> x = torch.rand(2, 2)
>>> linop.mm(x)
tensor([[0.1991, 0.1011],
        [0.1696, 0.0684],
```

(continues on next page)

(continued from previous page)

```
        [0.3345, 0.1180]])
>>> x = torch.rand(3, 2)
>>> linop.mv(x)
tensor([[0.6137, 0.3879, 0.6369],
        [0.7220, 0.5680, 1.0753],
        [0.7821, 0.5460, 0.9626]])
```

`__init__`(*mat: Tensor*, *is_hermitian: bool*) → None

Initialize the `MatrixLinearOperator`.

Parameters

- **`mat`** (*`torch.Tensor`*) – The matrix to be wrapped.
- **`is_hermitian`** (*`bool`*) – Indicating if the matrix is Hermitian. If `None`, the symmetry will be checked. If supplied as a bool, there is no check performed.

class `PureFunction`(*fcntocall: Callable*)

PureFunction class wraps methods to make it stateless and expose the pure function to take inputs of the original inputs (*params*) and the object's states (*objparams*). For functions, this class only acts as a thin wrapper.

Restore stack stores list of (objparams, identical) everytime the objparams are set, it will store the old objparams and indication if the old and new objparams are identical.

For Using this Class we first need to implement *_get_all_obj_params_init* and *_set_all_obj_params*.

Examples

```
>>> class WrapperFunction(PureFunction):
...     def _get_all_obj_params_init(self):
...         return []
...     def _set_all_obj_params(self, objparams):
...         pass
>>> def fcn(x, y):
...    return x + y
>>> pfunc = WrapperFunction(fcn)
>>> pfunc(1, 2)
3
```

`__init__`(*fcntocall: Callable*)

Initialize the PureFunction.

Parameters
`fcntocall` (*`Callable`*) – The function to be wrapped

`objparams`() → List

Get the current object parameters.

Returns
The current object parameters

Return type
List

`set_objparams`(*objparams: List*)

Set the object parameters.

Parameters

- **objparams** (*List*) – The object parameters to be set
- **TODO** (*check if identical with current object parameters*) –

restore_objparams()

Restore the object parameters to the previous state.

useobjparams(*objparams: List*)

Context manager to temporarily set the object parameters.

Parameters

objparams (*List*) – The object parameters to be set temporarily

disable_state_change()

Context manager to temporarily disable the state change.

class **FunctionPureFunction**(*fcntocall: Callable*)

Implementation of PureFunction for functions. It just acts as a thin wrapper for the function.

Examples

```
>>> def fcn(x, y):
...     return x + y
>>> pfunc = FunctionPureFunction(fcn)
>>> pfunc(1, 2)
3
```

class **EditableModulePureFunction**(*obj: EditableModule*, *method: Callable*)

Implementation of PureFunction for EditableModule.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import EditableModule, get_pure_
↪function
>>> class A(EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...     def mult(self, x):
...         return self.b * x
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
>>> B = A(4)
>>> m = get_pure_function(B.mult)
>>> m.set_objparams([3])
>>> m(2)
6
```

__init__(*obj: EditableModule*, *method: Callable*)

Initialize the EditableModulePureFunction.

Parameters

- **obj** (`EditableModule`) – The object to be wrapped
- **method** (*Callable*) – The method to be wrapped

class TorchNNPureFunction(*obj: Module*, *method: Callable*)

Implementation of PureFunction for torch.nn.Module.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import get_pure_function
>>> class A(torch.nn.Module):
...     def __init__(self, a):
...         super().__init__()
...         self.b = torch.nn.Parameter(torch.tensor(a*a))
...     def forward(self, x):
...         return self.b * x
>>> B = A(4.)
>>> m = get_pure_function(B.forward)
>>> m.set_objparams([3.])
>>> m(2)
6.0
```

__init__(*obj: Module*, *method: Callable*)

Initialize the TorchNNPureFunction.

Parameters

- **obj** (*torch.nn.Module*) – Object to be wrapped
- **method** (*Callable*) – Method to be wrapped

class PureFunction(*fcntocall: Callable*)

PureFunction class wraps methods to make it stateless and expose the pure function to take inputs of the original inputs (*params*) and the object's states (*objparams*). For functions, this class only acts as a thin wrapper.

Restore stack stores list of (objparams, identical) everytime the objparams are set, it will store the old objparams and indication if the old and new objparams are identical.

For Using this Class we first need to implement *_get_all_obj_params_init* and *_set_all_obj_params*.

Examples

```
>>> class WrapperFunction(PureFunction):
...     def _get_all_obj_params_init(self):
...         return []
...     def _set_all_obj_params(self, objparams):
...         pass
>>> def fcn(x, y):
...    return x + y
```

(continues on next page)

(continued from previous page)

```
>>> pfunc = WrapperFunction(fcn)
>>> pfunc(1, 2)
3
```

`__init__`(*fcntocall: Callable*)

Initialize the PureFunction.

Parameters
`fcntocall` (*`Callable`*) – The function to be wrapped

`objparams`() → List

Get the current object parameters.

Returns
The current object parameters

Return type
List

`set_objparams`(*objparams: List*)

Set the object parameters.

Parameters

- **`objparams`** (*`List`*) – The object parameters to be set
- **`TODO`** (*`check if identical with current object parameters`*) –

`restore_objparams`()

Restore the object parameters to the previous state.

`useobjparams`(*objparams: List*)

Context manager to temporarily set the object parameters.

Parameters
`objparams` (*`List`*) – The object parameters to be set temporarily

`disable_state_change`()

Context manager to temporarily disable the state change.

`_check_identical_objs`(*objs1: List*, *objs2: List*) → bool

Check if the two lists of objects are identical.

Examples

```
>>> l1 = [2, 2, 3]
>>> l2 = [1, 2, 3]
>>> _check_identical_objs(l1, l2)
False
```

Parameters

- **`objs1`** (*`List`*) – The first list of objects
- **`objs2`** (*`List`*) – The second list of objects

Returns
True if the two lists of objects are identical, False otherwise

Return type
bool

get_pure_function(*fcn*) → *PureFunction*

Get the pure function form of the function or method `fcn`.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import get_pure_function
>>> def fcn(x, y):
...     return x + y
>>> pfunc = get_pure_function(fcn)
>>> pfunc(1, 2)
3
```

Parameters
fcn (*function or method*) – Function or method to be converted into a `PureFunction` by exposing the hidden parameters affecting its outputs.

Returns
The pure function wrapper

Return type
PureFunction

set_default_option(*defopt: Dict*, *opt: Dict*) → Dict

return a dictionary based on the options and if no item from option, take it from defopt make a shallow copy to detach the results from defopt

Examples

```
>>> set_default_option({'a': 1, 'b': 2}, {'a': 3})
{'a': 3, 'b': 2}
```

Parameters

- **defopt** (*dict*) – Default options
- **opt** (*dict*) – Options

Returns
Merged options

Return type
dict

get_and_pop_keys(*dct: Dict*, *keys: List*) → Dict

Get and pop keys from a dictionary

Examples

```
>>> get_and_pop_keys({'a': 1, 'b': 2}, ['a'])
{'a': 1}
```

Parameters

- **dct** (*dict*) – Dictionary to pop from
- **keys** (*list*) – Keys to pop

Returns
Dictionary containing the popped keys

Return type
dict

get_method(*algname: str*, *methods: Mapping[str, Callable]*, *method: str | Callable*) → Callable

Get a method from a dictionary of methods

Examples

```
>>> get_method('foo', {'bar': lambda: 1}, 'bar')()
1
```

Parameters

- **algname** (*str*) – Name of the algorithm
- **methods** (*dict*) – Dictionary of methods
- **method** (*str or callable*) – Method to get

Returns
The method

Return type
callable

dummy_context_manager()

Dummy context manager

assert_runtime(*cond*, *msg=''*)

Assert at runtime

Examples

```
>>> assert_runtime(False, "This is a test")
Traceback (most recent call last):
...
RuntimeError: This is a test
```

Parameters

- **cond** (*bool*) – Condition to assert

- **msg** (*str*) – Message to raise if condition is not met

Raises
RuntimeError – If condition is not met

_set_initial_v(*vinit_type: str*, *dtype: dtype*, *device: device*, *batch_dims: Sequence*, *na: int*, *nguess: int*, *M: LinearOperator | None = None*) → Tensor

Set the initial guess for the eigenvectors.

Examples

```
>>> import torch
>>> vinit_type = "eye"
>>> dtype = torch.float64
>>> device = torch.device("cpu")
>>> batch_dims = (2, 3)
>>> na = 4
>>> nguess = 2
>>> M = None
>>> V = _set_initial_v(vinit_type, dtype, device, batch_dims, na, nguess, M)
>>> V
tensor([[[[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]],

         [[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]],

         [[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]]],


        [[[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]],

         [[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]],

         [[1., 0.],
          [0., 1.],
          [0., 0.],
          [0., 0.]]]], dtype=torch.float64)
```

Parameters

- **vinit_type** (*str*) – Mode of the initial guess ("randn", "rand", "eye")
- **dtype** (*torch.dtype*) – Data type of the initial guess.
- **device** (*torch.device*) – Device of the initial guess.
- **batch_dims** (*Sequence*) – Batch dimensions of the initial guess.
- **na** (*int*) – Number of basis functions.
- **nguess** (*int*) – Number of initial guesses.
- **M** (*Optional[LinearOperator] (default None)*) – The overlap matrix. If None, identity matrix is used.

Returns
    V – Initial guess for the eigenvectors.

Return type
    torch.Tensor

_take_eigpairs(*eival: Tensor*, *eivec: Tensor*, *neig: int*, *mode: str*)

Take the eigenpairs from the eigendecomposition.

Examples

```
>>> import torch
>>> eival = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
>>> eivec = torch.tensor([[[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]],
...                        [[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]]])
>>> neig = 2
>>> mode = "lowest"
>>> eival, eivec = _take_eigpairs(eival, eivec, neig, mode)
>>> eival
tensor([[1., 2.],
        [4., 5.]])
>>> eivec
tensor([[[1., 2.],
         [4., 5.],
         [7., 8.]],

        [[1., 2.],
         [4., 5.],
         [7., 8.]]])
```

Parameters

- **eival** (*torch.Tensor*) – Eigenvalues of the linear operator. Shape: `(*BV, na)`.
- **eivec** (*torch.Tensor*) – Eigenvectors of the linear operator. Shape: `(*BV, na, na)`.
- **neig** (*int*) – Number of eigenvalues and eigenvectors to be calculated.
- **mode** (*str*) – Mode of the eigenvalues to be calculated ("lowest", "uppest")

Returns

- **eival** (*torch.Tensor*) – Eigenvalues of the linear operator.
- **eivec** (*torch.Tensor*) – Eigenvectors of the linear operator.

exacteig(*A: LinearOperator*, *neig: int*, *mode: str*, *M: LinearOperator | None*) → Tuple[Tensor, Tensor]

Eigendecomposition using explicit matrix construction. No additional option for this method.

Examples

```
>>> import torch
>>> import numpy as np
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.rand(2, 2))
>>> neig = 2
>>> mode = "lowest"
>>> M = None
>>> evals, evecs = exacteig(A, neig, mode, M)
>>> evals.shape
torch.Size([2])
>>> evecs.shape
torch.Size([2, 2])
```

Parameters

- **A** (`LinearOperator`) – Linear operator to be diagonalized. Shape: `(*BA, q, q)`.
- **neig** (*int*) – Number of eigenvalues and eigenvectors to be calculated.
- **mode** (*str*) – Mode of the eigenvalues to be calculated (`"lowest"`, `"uppest"`)
- **M** (*Optional[LinearOperator] (default None)*) – The overlap matrix. If None, identity matrix is used. Shape: `(*BM, q, q)`.

Returns

- **evals** (*torch.Tensor*) – Eigenvalues of the linear operator.
- **evecs** (*torch.Tensor*) – Eigenvectors of the linear operator.

Warning:

- As this method construct the linear operators explicitly, it might requires a large memory.

degen_symeig(**args*, ***kwargs*)

A wrapper for torch.linalg.eigh to avoid complex eigenvalues for degenerate case.

Examples

```
>>> import torch
>>> import numpy as np
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.rand(2, 2))
>>> evals, evecs = degen_symeig.apply(A.fullmatrix())
>>> evals.shape
torch.Size([2])
>>> evecs.shape
torch.Size([2, 2])
```

davidson(*A: LinearOperator*, *neig: int*, *mode: str*, *M: LinearOperator | None = None*, *max_niter: int = 1000*, *nguess: int | None = None*, *v_init: str = 'randn'*, *max_addition: int | None = None*, *min_eps: float = 1e-06*, *verbose: bool = False*, ***unused*) → Tuple[Tensor, Tensor]

Using Davidson method for large sparse matrix eigendecomposition [2]_.

Examples

```
>>> import torch
>>> import numpy as np
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.rand(2, 2))
>>> neig = 2
>>> mode = "lowest"
>>> eigen_val, eigen_vec = davidson(A, neig, mode)
```

Parameters

- **A** (`LinearOperator`) – Linear operator to be diagonalized. Shape: `(*BA, q, q)`.
- **neig** (*int*) – Number of eigenvalues and eigenvectors to be calculated.
- **mode** (*str*) – Mode of the eigenvalues to be calculated (`"lowest"`, `"uppest"`)
- **M** (*Optional[LinearOperator] (default None)*) – The overlap matrix. If None, identity matrix is used. Shape: `(*BM, q, q)`.
- **max_niter** (*int*) – Maximum number of iterations
- **v_init** (*str*) – Mode of the initial guess (`"randn"`, `"rand"`, `"eye"`)
- **max_addition** (*int or None*) – Maximum number of new guesses to be added to the collected vectors. If None, set to `neig`.
- **min_eps** (*float*) – Minimum residual error to be stopped
- **verbose** (*bool*) – Option to be verbose

Returns

- **evals** (*torch.Tensor*) – Eigenvalues of the linear operator.
- **evecs** (*torch.Tensor*) – Eigenvectors of the linear operator.

References

lsymeig(*A: LinearOperator*, *neig: int | None = None*, *M: LinearOperator | None = None*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tuple[Tensor, Tensor]

Obtain `neig` lowest eigenvalues and eigenvectors of a linear operator

usymeig(*A: LinearOperator*, *neig: int | None = None*, *M: LinearOperator | None = None*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tuple[Tensor, Tensor]

Obtain `neig` uppest eigenvalues and eigenvectors of a linear operator

symeig(*A: LinearOperator*, *neig: int | None = None*, *mode: str = 'lowest'*, *M: LinearOperator | None = None*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tuple[Tensor, Tensor]

Obtain `neig` lowest eigenvalues and eigenvectors of a linear operator,

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.tensor([[3, -1j], [1j, 4]]))
>>> evals, evecs = symeig(A)
>>> evals.shape
torch.Size([2])
>>> evecs.shape
torch.Size([2, 2])
```

$$\mathbf{AX} = \mathbf{MXE}$$

where $\mathbf{A}$, $\mathbf{M}$ are linear operators, $\mathbf{E}$ is a diagonal matrix containing the eigenvalues, and $\mathbf{X}$ is a matrix containing the eigenvectors. This function can handle derivatives for degenerate cases by setting non-zero `degen_atol` and `degen_rtol` in the backward option using the expressions in [1]_.

Parameters

- **A** (`LinearOperator`) – The linear operator object on which the eigenpairs are constructed. It must be a Hermitian linear operator with shape `(*BA, q, q)`
- **neig** (*int or None*) – The number of eigenpairs to be retrieved. If `None`, all eigenpairs are retrieved
- **mode** (*str*) – `"lowest"` or `"uppermost"`/`"uppest"`. If `"lowest"`, it will take the lowest `neig` eigenpairs. If `"uppest"`, it will take the uppermost `neig`.
- **M** (`LinearOperator`) – The transformation on the right hand side. If `None`, then `M=I`. If specified, it must be a Hermitian with shape `(*BM, q, q)`.
- **bck_options** (*dict*) – Method-specific options for `solve()` which used in backpropagation calculation with some additional arguments for computing the backward derivatives:
  - `degen_atol` (`float` or None): Minimum absolute difference between two eigenvalues to be treated as degenerate. If None, it is `torch.finfo(dtype).eps**0.6`. If 0.0, no special treatment on degeneracy is applied. (default: None)
  - `degen_rtol` (`float` or None): Minimum relative difference between two eigenvalues to be treated as degenerate. If None, it is `torch.finfo(dtype).eps**0.4`. If 0.0, no special treatment on degeneracy is applied. (default: None)

  Note: the default values of `degen_atol` and `degen_rtol` are going to change in the future. So, for future compatibility, please specify the specific values.
- **method** (*str or callable or None*) – Method for the eigendecomposition. If `None`, it will choose `"exacteig"`.
- ****fwd_options** – Method-specific options (see method section below).

Returns
It will return eigenvalues and eigenvectors with shapes respectively `(*BAM, neig)` and `(*BAM, na, neig)`, where `*BAM` is the broadcasted shape of `*BA` and `*BM`.

Return type
tuple of tensors (eigenvalues, eigenvectors)

References

class **symeig_torchfcn**(**args*, ***kwargs*)

A wrapper for symeig to be used in torch.autograd.Function

static **forward**(*ctx*, *A*, *neig*, *mode*, *M*, *fwd_options*, *bck_options*, *na*, **amparams*)

Calculate the eigenvalues and eigenvectors of a linear operator

Parameters

- **A** (`LinearOperator`) – The linear operator object on which the eigenpairs are constructed. It must be a Hermitian linear operator with shape `(*BA, q, q)`
- **neig** (*int*) – The number of eigenpairs to be retrieved. If `None`, all eigenpairs are retrieved
- **mode** (*str*) – `"lowest"` or `"uppermost"`/`"uppest"`. If `"lowest"`, it will take the lowest `neig` eigenpairs. If `"uppest"`, it will take the uppermost `neig`.
- **M** (*xitorch.LinearOperator*) – The transformation on the right hand side. If `None`, then `M=I`. If specified, it must be a Hermitian with shape `(*BM, q, q)`.
- **fwd_options** (*dict*) – Method-specific options (see method section below).
- **bck_options** (*dict*) – Method-specific options for `solve()` which used in backpropagation calculation with some additional arguments for computing the backward derivatives: `degen_atol` and `degen_rtol`.
- **na** (*int*) – Number of parameters of A (and M if M is not None)
- ***amparams** (*torch.Tensor*) – Parameters of A (and M if M is not None)

static **backward**(*ctx*, *grad_evals*, *grad_evecs*)

Calculate the gradient of the eigenvalues and eigenvectors of a linear operator

Parameters

- **grad_evals** (*torch.Tensor*) – The gradient of the eigenvalues. Shape: `(*BAM, neig)`
- **grad_evecs** (*torch.Tensor*) – The gradient of the eigenvectors. Shape: `(*BAM, na, neig)`

_check_degen(*evals: Tensor*, *degen_atol: float*, *degen_rtol: float*) → Tuple[Tensor, bool]

Check the degeneracy of the eigenvalues

Examples

```
>>> import torch
>>> evals = torch.tensor([1, 1, 2, 3, 3, 3, 4, 5, 5])
>>> degen_atol = 0.1
>>> degen_rtol = 0.1
>>> idx_degen, isdegenerate = _check_degen(evals, degen_atol, degen_rtol)
>>> idx_degen.shape
torch.Size([9, 9])
>>> isdegenerate
True
```

Parameters

- **evals** (*torch.Tensor*) – Eigenvalues of the linear operator. Shape: `(*BAM, neig)`

- **degen_atol** (*float*) – Minimum absolute difference between two eigenvalues to be treated as degenerate.
- **degen_rtol** (*float*) – Minimum relative difference between two eigenvalues to be treated as degenerate.

Returns

- **idx_degen** (*torch.Tensor*) – The degeneracy map. Shape: `(*BAM, neig, neig)`
- **isdegenerate** (*bool*) – Whether the eigenvalues are degenerate

`ortho`(*A: Tensor*, *B: Tensor*, ***, *D: Tensor | None = None*, *M: LinearOperator | None = None*, *mright: bool = False*) → Tensor

Orthogonalize A w.r.t. B

Examples

```
>>> import torch
>>> A = torch.tensor([[1, 2], [3, 4]])
>>> B = torch.tensor([[1, 0], [0, 1]])
>>> ortho(A, B)
tensor([[0, 2],
        [3, 0]])
```

Parameters

- **A** (*torch.Tensor*) – The tensor to be orthogonalized. Shape: `(*BAM, na, neig)`
- **B** (*torch.Tensor*) – The tensor to be orthogonalized against. Shape: `(*BAM, na, neig)`
- **D** (*torch.Tensor or None*) – The degeneracy map. If None, it is identity matrix. Shape: `(*BAM, neig, neig)`
- **M** (*LinearOperator or None*) – The overlap matrix. If None, identity matrix is used. Shape: `(*BM, q, q)`
- **mright** (*bool*) – Whether to operate M at the right or at the left

Returns
The orthogonalized tensor. Shape: `(*BAM, na, neig)`

Return type
torch.Tensor

`jac`(*fcn: Callable[[...], Tensor]*, *params: Sequence[Any]*, *idxs: None | int | Sequence[int] = None*)

Returns the LinearOperator that acts as the jacobian of the params. The shape of LinearOperator is (nout, nin) where *nout* and *nin* are the total number of elements in the output and the input, respectively.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import jac
>>> def fcn(x, y):
...     return x * y
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> y = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)
>>> jac(fcn, [x, y])
[LinearOperator (_Jac) with shape (3, 3), dtype = torch.float32, device = cpu,␣
↪LinearOperator (_Jac) with shape (3, 3), dtype = torch.float32, device = cpu]
```

Parameters

- **`fcn`** (*`Callable[...,torch.Tensor]`*) – Callable with tensor output and arbitrary numbers of input parameters.
- **`params`** (*`Sequence[Any]`*) – List of input parameters of the function.
- **`idxs`** (*`int or list of int or None`*) – List of the parameters indices to get the jacobian. The pointed parameters in *params* must be tensors and requires_grad. If it is None, then it will return all jacobian for all parameters that are tensor which requires_grad.

Returns
linops – List of LinearOperator of the jacobian

Return type
Union[*LinearOperator*, List]

`class _Jac`(**args*, ***kwargs*)

Jacobian of a function with respect to a parameter in the function.

Examples

```
>>> import torch
>>> def fcn(x, y):
...     return x * y
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> y = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)
>>> pfcn = get_pure_function(fcn)
>>> _Jac(pfcn, [x, y], 1)
LinearOperator (_Jac) with shape (3, 3), dtype = torch.float32, device = cpu
```

`__init__`(*fcn: PureFunction*, *params: Sequence[Any]*, *idx: int*, *is_hermitian=False*) → None

Initialize the _Jac object.

Parameters

- **`fcn`** (`PureFunction`) – The function that will be differentiated.
- **`params`** (*`Sequence[Any]`*) – List of input parameters of the function.
- **`idx`** (*`int`*) – The index of the parameter to be differentiated.
- **`is_hermitian`** (*`bool`*) – If True, then the LinearOperator is hermitian.

_setup_idxs(*idxs: None | int | Sequence[int]*, *params: Sequence[Any]*) → Sequence[int]

Check the idxs and return the list of indices.

Examples

```
>>> import torch
>>> import numpy as np
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> y = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)
>>> _setup_idxs(None, [x, y])
[0, 1]
```

Parameters

- **idxs** (*int or list of int or None*) – List of the parameters indices to get the jacobian. The pointed parameters in *params* must be tensors and requires_grad. If it is None, then it will return all jacobian for all parameters that are tensor which requires_grad.
- **params** (*Sequence[Any]*) – List of input parameters of the function.

Returns

idxs – List of the parameters indices to get the jacobian.

Return type

list of int

connect_graph(*out: Tensor*, *params: Sequence[Any]*)

Just to have a dummy graph, in case there is a parameter that is disconnected in calculating df/dy.

Examples

```
>>> import torch
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> y = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)
>>> out = x * y
>>> connect_graph(out, [x, y])
tensor([ 4., 10., 18.], grad_fn=<AddBackward0>)
```

Parameters

- **out** (*torch.Tensor*) – The output tensor. It will be added with a dummy graph.
- **params** (*Sequence[Any]*) – List of parameters that will be added with a dummy graph.

Returns

out – The output tensor with a dummy graph.

Return type

torch.Tensor

wrap_gmres(*A*, *B*, *E=None*, *M=None*, *min_eps=1e-09*, *max_niter=None*, ***unused*)

Using SciPy's gmres method to solve the linear equation.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.tensor([[1., 2], [3, 4]]))
>>> B = torch.tensor([[[5., 6], [7, 8]]])
>>> wrap_gmres(A, B, None, None)
tensor([[[-3.0000, -4.0000],
         [ 4.0000,  5.0000]]])
```

Parameters

- **A** (`LinearOperator`) – The linear operator A to be solved. Shape: (*BA*, na, na)
- **B** (*torch.Tensor*) – Batched matrix B. Shape: (*BB*, na, ncols)
- **E** (*torch.Tensor or None*) – Batched vector E. Shape: (*BE*, ncols)
- **M** (`LinearOperator` *or None*) – The linear operator M. Shape: (*BM*, na, na)
- **min_eps** (*float*) – Relative tolerance for stopping conditions
- **max_niter** (*int or None*) – Maximum number of iterations. If `None`, default to twice of the number of columns of `A`.

Returns
The Solution matrix X. Shape: (*BBE*, na, ncols)

Return type
torch.Tensor

`exactsolve`(*A:* LinearOperator, *B: Tensor*, *E: Tensor* | *None*, *M:* LinearOperator | *None*)

Solve the linear equation by contructing the full matrix of LinearOperators.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.tensor([[1., 2], [3, 4]]))
>>> B = torch.tensor([[5., 6], [7, 8]])
>>> exactsolve(A, B, None, None)
tensor([[-3., -4.],
        [ 4.,  5.]])
```

Parameters

- **A** (`LinearOperator`) – The linear operator A to be solved. Shape: (*BA*, na, na)
- **B** (*torch.Tensor*) – Batched matrix B. Shape: (*BB*, na, ncols)
- **E** (*torch.Tensor or None*) – Batched vector E. Shape: (*BE*, ncols)
- **M** (`LinearOperator` *or None*) – The linear operator M. Shape: (*BM*, na, na)

Returns
The Solution matrix X. Shape: (*BBE*, na, ncols)

Return type
torch.Tensor

Warning:

- As this method construct the linear operators explicitly, it might requires a large memory.

solve_ABE(*A: Tensor*, *B: Tensor*, *E: Tensor*)

Solve the linear equation AX = B - diag(E)X.

Examples

```
>>> import torch
>>> A = torch.tensor([[1., 2], [3, 4]])
>>> B = torch.tensor([[5., 6], [7, 8]])
>>> E = torch.tensor([1., 2])
>>> solve_ABE(A, B, E)
tensor([[-0.1667,  0.5000],
        [ 2.5000,  3.2500]])
```

Parameters

- **A** (*torch.Tensor*) – The batched matrix A. Shape: (**BA*, na, na)
- **B** (*torch.Tensor*) – The batched matrix B. Shape: (**BB*, na, ncols)
- **E** (*torch.Tensor*) – The batched vector E. Shape: (**BE*, ncols)

Returns
The batched matrix X.

Return type
torch.Tensor

get_batchdims(*A:* LinearOperator, *B: Tensor*, *E: Tensor* | *None*, *M:* LinearOperator | *None*)

Get the batch dimensions of the linear operator and the matrix B

Examples

```
>>> from deepchem.utils.differentiation_utils import MatrixLinearOperator
>>> import torch
>>> A = MatrixLinearOperator(torch.randn(4, 3, 3), True)
>>> B = torch.randn(3, 3, 2)
>>> get_batchdims(A, B, None, None)
[4]
```

Parameters

- **A** (LinearOperator) – The linear operator. It can be a batched linear operator.
- **B** (*torch.Tensor*) – The matrix B. It can be a batched matrix.
- **E** (*Union[torch.Tensor, None]*) – The matrix E. It can be a batched matrix.

- **M** (*Union*[LinearOperator, *None*]) – The linear operator M. It can be a batched linear operator.

Returns
The batch dimensions of the linear operator and the matrix B

Return type
List[int]

setup_precond(*precond: LinearOperator | None = None*) → Callable[[Tensor], Tensor]

Setup the preconditioning function

Examples

```
>>> from deepchem.utils.differentiation_utils import MatrixLinearOperator
>>> import torch
>>> A = MatrixLinearOperator(torch.randn(4, 3, 3), True)
>>> B = torch.randn(4, 3, 2)
>>> cond = setup_precond(A)
>>> cond(B).shape
torch.Size([4, 3, 2])
```

Parameters
precond (*Optional*[LinearOperator]) – The preconditioning linear operator. If None, no preconditioning is applied.

Returns
The preconditioning function. It takes a tensor and returns a tensor.

Return type
Callable[[torch.Tensor], torch.Tensor]

dot(*r: Tensor*, *z: Tensor*) → Tensor

Dot product of two vectors. r and z must have the same shape. Then sums it up across the last dimension.

Examples

```
>>> import torch
>>> r = torch.tensor([[1, 2], [3, 4]])
>>> z = torch.tensor([[5, 6], [7, 8]])
>>> dot(r, z)
tensor([[26, 44]])
```

Parameters

- **r** (*torch.Tensor*) – The first vector. Shape: (**BR*, nr, nc)
- **z** (*torch.Tensor*) – The second vector. Shape: (**BR*, nr, nc)

Returns
The dot product of r and z. Shape: (**BR*, 1, nc)

Return type
torch.Tensor

gmres(*A: LinearOperator*, *B: Tensor*, *E: Tensor | None = None*, *M: LinearOperator | None = None*, *posdef: bool | None = None*, *max_niter: int | None = None*, *rtol: float = 1e-06*, *atol: float = 1e-08*, *eps: float = 1e-12*, ***unused*) → Tensor

Solve the linear equations using Generalised minial residual method.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.tensor([[1., 2], [3, 4]]))
>>> B = torch.tensor([[5., 6], [7, 8]])
>>> gmres(A, B)
tensor([[0.8959, 1.0697],
        [1.2543, 1.4263]])
```

Parameters

- **A** (`LinearOperator`) – The linear operator A to be solved. Shape: (**BA*, na, na)
- **B** (*torch.Tensor*) – Batched matrix B. Shape: (**BB*, na, ncols)
- **E** (*torch.Tensor or None*) – Batched vector E. Shape: (**BE*, ncols)
- **M** (`LinearOperator` *or None*) – The linear operator M. Shape: (**BM*, na, na)
- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.

Returns
The solution matrix X. Shape: (**BBE*, na, ncols)

Return type
torch.Tensor

setup_linear_problem(*A: LinearOperator*, *B: Tensor*, *E: Tensor | None*, *M: LinearOperator | None*, *batchdims: Sequence[int]*, *posdef: bool | None*, *need_hermit: bool*) → Tuple[Callable[[Tensor], Tensor], Callable[[Tensor], Tensor], Tensor, bool]

Setup the linear problem for solving AX = B

Examples

```
>>> from deepchem.utils.differentiation_utils import MatrixLinearOperator
>>> import torch
>>> A = MatrixLinearOperator(torch.randn(4, 3, 3), True)
>>> B = torch.randn(4, 3, 2)
>>> A_fcn, AT_fcn, B_new, col_swapped = setup_linear_problem(A, B, None, None, [4],␣
↪None, False)
>>> A_fcn(B).shape
torch.Size([4, 3, 2])
```

Parameters

- **A** (LinearOperator) – The linear operator A. It can be a batched linear operator.
- **B** (*torch.Tensor*) – The matrix B. It can be a batched matrix.
- **E** (*Optional[torch.Tensor]*) – The matrix E. It can be a batched matrix.
- **M** (*Optional[*LinearOperator*]*) – The linear operator M. It can be a batched linear operator.
- **batchdims** (*Sequence[int]*) – The batch dimensions of the linear operator and the matrix B
- **posdef** (*Optional[bool]*) – Whether the linear operator is positive definite. If None, it will be estimated.
- **need_hermit** (*bool*) – Whether the linear operator is Hermitian. If True, it will be estimated.

Returns

Callable[[torch.Tensor], torch.Tensor],
torch.Tensor, bool]

The function A, its transposed function, the matrix B, and whether the columns of B are swapped.

Return type

Tuple[Callable[[torch.Tensor], torch.Tensor],

safedenom(*r: Tensor*, *eps: float*) → Tensor

Make sure the denominator is not zero

Examples

```
>>> import torch
>>> r = torch.tensor([[0., 2], [3, 4]])
>>> safedenom(r, 1e-9)
tensor([[1.0000e-09, 2.0000e+00],
        [3.0000e+00, 4.0000e+00]])
```

Parameters

- **r** (*torch.Tensor*) – The input tensor. Shape: (**BR*, nr, nc)
- **eps** (*float*) – The small number to replace the zero denominator

Returns
The tensor with non-zero denominator. Shape: (*BR*, nr, nc)

Return type
torch.Tensor

`get_largest_eival`(*Afcn: Callable*, *x: Tensor*) → Tensor

Get the largest eigenvalue of the linear operator Afcn

Examples

```
>>> import torch
>>> def Afcn(x):
...     return 10 * x
>>> x = torch.tensor([[1., 2], [3, 4]])
>>> get_largest_eival(Afcn, x)
tensor([[10., 10.]])
```

Parameters

- **Afcn** (*Callable*) – The linear operator A. It takes a tensor and returns a tensor.
- **x** (*torch.Tensor*) – The input tensor. Shape: (*, nr, nc)

Returns
The largest eigenvalue. Shape: (*, 1, nc)

Return type
torch.Tensor

`solve`(*A: LinearOperator*, *B: Tensor*, *E: Tensor | None = None*, *M: LinearOperator | None = None*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tensor

Performing iterative method to solve the equation.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils import LinearOperator
>>> A = LinearOperator.m(torch.tensor([[1., 2], [3, 4]]))
>>> B = torch.tensor([[5., 6], [7, 8]])
>>> solve(A, B)
tensor([[-3., -4.],
        [ 4.,  5.]])
```

$$\mathbf{AX} = \mathbf{B}$$

or

$$\mathbf{AX} - \mathbf{MXE} = \mathbf{B}$$

where $\mathbf{E}$ is a diagonal matrix. This function can also solve batched multiple inverse equation at the same time by applying $\mathbf{A}$ to a tensor $\mathbf{X}$ with shape `(...,na,ncols)`. The applied $\mathbf{E}$ are not necessarily identical for each column.

Parameters

- **A** (LinearOperator) – A linear operator that takes an input X and produce the vectors in the same space as B. It should have the shape of (*BA, na, na)
- **B** (*torch.Tensor*) – The tensor on the right hand side with shape (*BB, na, ncols)
- **E** (*Union[torch.Tensor, None]*) – If a tensor, it will solve $\mathbf{AX}-\mathbf{MXE}=\mathbf{B}$. It will be regarded as the diagonal of the matrix. Otherwise, it just solves $\mathbf{AX}=\mathbf{B}$ and M is ignored. If it is a tensor, it should have shape of (*BE, ncols).
- **M** (*Optional[*LinearOperator*]*) – The transformation on the E side. If E is None, then this argument is ignored. If E is not None and M is None, then M=I. If LinearOperator, it must be Hermitian with shape (*BM, na, na).
- **bck_options** (*dict*) – Options of the iterative solver in the backward calculation.
- **method** (*Union[str, Callable, None]*) – The method of linear equation solver. If None, it will choose "cg" or "bicgstab" based on the matrices symmetry. *Note*: default method will be changed quite frequently, so if you want future compatibility, please specify a method.
- ****fwd_options** – Method-specific options

Returns
The tensor $\mathbf{X}$ that satisfies $\mathbf{AX}-\mathbf{MXE}=\mathbf{B}$.

Return type
torch.Tensor

broyden1_solve(*fcn: Callable*, *x0: Tensor*, *params*, *method: str*, *alpha=None*, *uv0=None*, *max_rank=None*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *line_search=True*, *verbose=False*, *custom_terminator=None*, ***unused*)

Solve the root finder or linear equation using the first Broyden method [1]_. It can be used to solve minimization by finding the root of the function's gradient.

Examples

```
>>> def fcn(x):
...     return x**2 - 4
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = broyden1(fcn, x0)
>>> x
tensor(-2.0000, grad_fn=<ViewBackward0>)
```

Parameters

- **fcn** (*callable*) – The function to solve. It should take a tensor and return a tensor.
- **x0** (*torch.Tensor*) – The initial guess of the solution.
- **params** (*tuple*) – The parameters to pass to the function.

References

_rootfinder_solve(*alg: str*, *A:* LinearOperator, *B: Tensor*, *E: Tensor* | *None = None*, *M:* LinearOperator | *None = None*, ***options*)

Solve the linear equations using rootfinder algorithm

Examples

```
>>> import torch
>>> A = torch.tensor([[1., 2], [3, 4]])
>>> B = torch.tensor([[5., 6], [7, 8]])
>>> _rootfinder_solve("broyden1", A, B)
tensor([[-3.0000, -4.0000],
        [ 4.0000,  5.0000]])
```

Parameters

- **alg** (*str*) – The algorithm to use. Currently, only "broyden1" is supported.
- **A** (*torch.Tensor*) – The matrix A. Shape: (**BA*, nr, nr)
- **B** (*torch.Tensor*) – The matrix B. Shape: (**BB*, nr, ncols)
- **E** (*torch.Tensor or None*) – The matrix E. Shape: (**BE*, ncols)
- **M** (*torch.Tensor or None*) – The matrix M. Shape: (**BM*, nr, nr)
- **options** (*dict*) – The options for the rootfinder algorithm

Returns

The solution matrix X. Shape: (**BBE*, nr, ncols)

Return type

torch.Tensor

cg(*A:* LinearOperator, *B: Tensor*, *E: Tensor* | *None = None*, *M:* LinearOperator | *None = None*, *posdef: bool* | *None = None*, *precond:* LinearOperator | *None = None*, *max_niter: int* | *None = None*, *rtol: float = 1e-06*, *atol: float = 1e-08*, *eps: float = 1e-12*, *resid_calc_every: int = 10*, *verbose: bool = False*, ***unused*) → Tensor

Solve the linear equations using Conjugate-Gradient (CG) method.

Parameters

- **A** (LinearOperator) – A linear operator that takes an input `X` and produce the vectors in the same space as `B`. It should have the shape of `(*BA, na, na)`
- **B** (*torch.Tensor*) – The tensor on the right hand side with shape `(*BB, na, ncols)`
- **E** (*Union[torch.Tensor, None]*) – If a tensor, it will solve $\mathbf{AX} - \mathbf{MXE} = \mathbf{B}$. It will be regarded as the diagonal of the matrix. Otherwise, it just solves $\mathbf{AX} = \mathbf{B}$ and `M` is ignored. If it is a tensor, it should have shape of `(*BE, ncols)`.
- **M** (*Optional[*LinearOperator*]*) – The transformation on the `E` side. If `E` is `None`, then this argument is ignored. If E is not `None` and `M` is `None`, then `M=I`.
- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **precond** (LinearOperator *or None*) – LinearOperator for the preconditioning. If None, no preconditioner is applied.

- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.
- **resid_calc_every** (*int*) – Calculate the residual in its actual form instead of substitution form with this frequency, to avoid rounding error accummulation. If your linear operator has bad numerical precision, set this to be low. If 0, then never calculate the residual in its actual form.
- **verbose** (*bool*) – Verbosity of the algorithm.

bicgstab(*A: LinearOperator*, *B: Tensor*, *E: Tensor | None = None*, *M: LinearOperator | None = None*, *posdef: bool | None = None*, *precond_l: LinearOperator | None = None*, *precond_r: LinearOperator | None = None*, *max_niter: int | None = None*, *rtol: float = 1e-06*, *atol: float = 1e-08*, *eps: float = 1e-12*, *verbose: bool = False*, *resid_calc_every: int = 10*, ***unused*) → Tensor

Solve the linear equations using stabilized Biconjugate-Gradient method.

Parameters

- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **precond_l** (*LinearOperator or None*) – LinearOperator for the left preconditioning. If None, no preconditioner is applied.
- **precond_r** (*LinearOperator or None*) – LinearOperator for the right preconditioning. If None, no preconditioner is applied.
- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.
- **resid_calc_every** (*int*) – Calculate the residual in its actual form instead of substitution form with this frequency, to avoid rounding error accummulation. If your linear operator has bad numerical precision, set this to be low. If 0, then never calculate the residual in its actual form.
- **verbose** (*bool*) – Verbosity of the algorithm.

class solve_torchfcn(**args*, ***kwargs*)

static forward(*ctx*, *A*, *B*, *E*, *M*, *method*, *fwd_options*, *bck_options*, *na*, **all_params*)

Forward calculation of the solve function.

Parameters

- **A** (*LinearOperator*) – A linear operator that takes an input `X` and produce the vectors in the same space as `B`. It should have the shape of `(*BA, na, na)`
- **B** (*torch.Tensor*) – The tensor on the right hand side with shape `(*BB, na, ncols)`

- **E** (*Union[torch.Tensor, None]*) – If a tensor, it will solve $\mathbf{AX} - \mathbf{MXE} = \mathbf{B}$. It will be regarded as the diagonal of the matrix. Otherwise, it just solves $\mathbf{AX} = \mathbf{B}$ and `M` is ignored. If it is a tensor, it should have shape of `(*BE, ncols)`.
- **M** (*Optional[LinearOperator]*) – The transformation on the `E` side. If `E` is `None`, then this argument is ignored. If E is not `None` and `M` is `None`, then `M=I`.
- **method** (*Union[str, Callable, None]*) – The method of linear equation solver. If `None`, it will choose `"cg"` or `"bicgstab"` based on the matrices symmetry. *Note*: default method will be changed quite frequently, so if you want future compatibility, please specify a method.
- **fwd_options** – Method-specific options
- **bck_options** (*dict*) – Options of the iterative solver in the backward calculation.
- **na** (*int*) – Number of parameters of A
- **all_params** (*Sequence[torch.Tensor]*) – All the parameters of M and A

static backward(*ctx*, *grad_x*)

Define a formula for differentiating the operation with backward mode automatic differentiation.

This function is to be overridden by all subclasses. (Defining this function is equivalent to defining the `vjp` function.)

It must accept a context `ctx` as the first argument, followed by as many outputs as the *forward()* returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to *forward()*. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., *backward()* will have `ctx.needs_input_grad[0] = True` if the first input to *forward()* needs gradient computed w.r.t. the output.

anderson_acc(*fcn: Callable[[...], Tensor]*, *x0: Tensor*, *params: List*, *feat_ndims: int = 1*, *msize: int = 5*, *beta: float = 1.0*, *lmbda: float = 0.0001*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *custom_terminator=None*, *verbose: bool = False*) → Tensor

Solve the equilibrium (or fixed-point iteration) problem using Anderson acceleration.

Examples

```
>>> import torch
>>> def fcn(x):
...     return x
>>> x0 = torch.tensor([0.0], requires_grad=True)
>>> x = anderson_acc(fcn, x0, [], 2, 10, maxiter=1000)
>>> x
tensor([0.], requires_grad=True)
```

Parameters

- **feat_ndims** (*int*) – The number of dimensions at the end that describe the features (i.e. non-batch dimensions)
- **msize** (*int*) – The maximum number of previous iterations we should save for the algorithm

- **beta** (*float*) – The damped or overcompensated parameters
- **lmbda** (*float*) – Small number to ensure invertability of the matrix
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f - x`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f - x`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **verbose** (*bool*) – Options for verbosity

References

gd(*fcn: Callable[[...], Tensor], x0: Tensor, params: List, step: float = 0.001, gamma: float = 0.9, maxiter: int = 1000, f_tol: float = 0.0, f_rtol: float = 1e-08, x_tol: float = 0.0, x_rtol: float = 1e-08, verbose=False, **unused*)

Vanilla gradient descent with momentum. The stopping conditions use OR criteria. The update step is following the equations below.

Examples

```
>>> import torch
>>> from deepchem.utils.differentiation_utils.optimize.minimizer import gd
>>> def fcn(x):
...     return (x - 2) ** 2, 2 * (x - 2)
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = gd(fcn, x0, [])
>>> x
tensor(2.0000)
```

$$\mathbf{v}_{t+1} = \gamma \mathbf{v}_t - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_t)$$
$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1}$$

Parameters

- **fcn** (*callable*) – The objective function to minimize. It should take a tensor and return a tensor and its gradient.
- **x0** (*torch.Tensor*) – The initial guess.
- **step** (*float*) – The step size towards the steepest descent direction, i.e. η in the equations above.
- **gamma** (*float*) – The momentum factor, i.e. γ in the equations above.
- **maxiter** (*int*) – Maximum number of iterations.
- **f_tol** (*float or None*) – The absolute tolerance of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.

adam(*fcn: Callable[[...], Tensor]*, *x0: Tensor*, *params: List*, *step: float = 0.001*, *beta1: float = 0.9*, *beta2: float = 0.999*, *eps: float = 1e-08*, *maxiter: int = 1000*, *f_tol: float = 0.0*, *f_rtol: float = 1e-08*, *x_tol: float = 0.0*, *x_rtol: float = 1e-08*, *verbose=False*, ***unused*)

Adam optimizer by Kingma & Ba (2015). The stopping conditions use OR criteria. The update step is following the equations below.

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.minimizer import adam
>>> def fcn(x):
...     return (x - 4) * 2, (x * 2) + 3
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = adam(fcn, x0, [], maxiter=10000)
>>> x
tensor(-1.4999)
```

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_{\mathbf{x}} f(\mathbf{x}_{t-1}) \\
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
\mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
\hat{\mathbf{m}}_t &= \mathbf{m}_t/(1-\beta_1^t) \\
\hat{\mathbf{v}}_t &= \mathbf{v}_t/(1-\beta_2^t) \\
\mathbf{x}_t &= \mathbf{x}_{t-1} - \alpha\hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)
\end{aligned}
$$

Parameters

- **step** (*float*) – The step size towards the descent direction, i.e. α in the equations above.
- **beta1** (*float*) – Exponential decay rate for the first moment estimate.
- **beta2** (*float*) – Exponential decay rate for the first moment estimate.
- **eps** (*float*) – Small number to prevent division by 0.
- **maxiter** (*int*) – Maximum number of iterations.
- **f_tol** (*float or None*) – The absolute tolerance of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.

TerminationCondition(*f_tol: float*, *f_rtol: float*, *x_tol: float*, *x_rtol: float*, *verbose: bool*)

The class to handle the stopping conditions.

Examples

```
>>> stop_cond = TerminationCondition(1e-8, 1e-8, 1e-8, 1e-8, True)
```

_nonlin_solver(*fcn: Callable*, *x0: Tensor*, *params*, *method: str*, *alpha=None*, *uv0=None*, *max_rank=None*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *line_search=True*, *verbose=False*, *custom_terminator=None*, ***unused*)

Parameters

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is - alpha * I + u v^T.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is - alpha * I + u v^T. If "svd", then it uses 1-rank svd to obtain u and v. If None, then u and v are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If None, it is inf.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output f.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output f.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input x.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input x.
- **line_search** (*bool or str*) – Options to perform line search. If True, it is set to "armijo".
- **verbose** (*bool*) – Options for verbosity

broyden1(*fcn: Callable*, *x0: Tensor*, *params*, *method: str*, *alpha=None*, *uv0=None*, *max_rank=None*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *line_search=True*, *verbose=False*, *custom_terminator=None*, ***unused*)

Solve the root finder or linear equation using the first Broyden method [1]_. It can be used to solve minimization by finding the root of the function's gradient.

Examples

```
>>> def fcn(x):
...     return x**2 - 4
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = broyden1(fcn, x0)
>>> x
tensor(-2.0000, grad_fn=<ViewBackward0>)
```

Parameters

- **fcn** (*callable*) – The function to solve. It should take a tensor and return a tensor.
- **x0** (*torch.Tensor*) – The initial guess of the solution.
- **params** (*tuple*) – The parameters to pass to the function.

References

broyden2(*fcn: Callable*, *x0: Tensor*, *params*, *method: str*, *alpha=None*, *uv0=None*, *max_rank=None*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *line_search=True*, *verbose=False*, *custom_terminator=None*, ***unused*)

Solve the root finder or linear equation using the second Broyden method [2]_. It can be used to solve minimization by finding the root of the function's gradient.

Examples

```
>>> def fcn(x):
...     return x**2 - 4
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = broyden1(fcn, x0)
>>> x
tensor(-2.0000, grad_fn=<ViewBackward0>)
```

Parameters

- **fcn** (*callable*) – The function to solve. It should take a tensor and return a tensor.
- **x0** (*torch.Tensor*) – The initial guess of the solution.
- **params** (*tuple*) – The parameters to pass to the function.

References

linearmixing(*fcn: Callable*, *x0: Tensor*, *params=()*, *alpha=None*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *line_search=True*, *verbose=False*, ***unused*)

Solve the root finding problem by approximating the inverse of Jacobian to be a constant scalar.

Examples

```
>>> def fcn(x):
...     return x**2 - 4
>>> x0 = torch.tensor(0.0, requires_grad=True)
>>> x = broyden1(fcn, x0)
>>> x
tensor(-2.0000, grad_fn=<ViewBackward0>)
```

Parameters

- **fcn** (*Callable*) – The function to solve. It should take a tensor and return a tensor.
- **x0** (*torch.Tensor*) – The initial guess of the solution.
- **params** (*tuple*) – The parameters to pass to the function.
- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `-alpha * I`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

References

`_safe_norm`(*v*)

Compute the norm of a vector, checking for finite values.

`_nonline_line_search`(*func: Callable*, *x: Tensor*, *y: Tensor*, *dx: Tensor*, *search_type='armijo'*, *rdiff=1e-08*, *smin=0.01*)

Find a suitable step length for a line search.

Parameters

- **func** (*Callable*) – The function to minimize.
- **x** (*torch.Tensor*) – The current point.
- **y** (*torch.Tensor*) – The function value at the current point.
- **dx** (*torch.Tensor*) – The search direction.
- **search_type** (*str*) – The type of line search to perform. Currently, only "armijo" is supported.
- **rdiff** (*float*) – The relative difference to compute the derivative.
- **smin** (*float*) – The minimum step length to take.

Returns

- **s** (*float*) – The step length.
- **x** (*torch.Tensor*) – The new point.
- **y** (*torch.Tensor*) – The function value at the new point.
- **y_norm** (*float*) – The norm of the function value at the new point.

`_scalar_search_armijo`(*phi: Callable*, *phi0: float*, *derphi0: float*, *c1: float = 0.0001*, *alpha0=1*, *amin=0*, *max_niter=20*)

Minimize over alpha, the function phi(s) at the current point and the derivative derphi(s) at the current point.

Parameters

- **phi** (*callable*) – The function to minimize.
- **phi0** (*float*) – The value of phi at 0.
- **derphi0** (*float*) – The value of the derivative of phi at 0.
- **c1** (*float*) – The Armijo condition parameter.
- **alpha0** (*float*) – The initial guess of the step length.
- **amin** (*float*) – The minimum step length to take.
- **max_niter** (*int*) – The maximum number of iterations to take.

Returns

- **alpha** (*float*) – The step length.
- **phi** (*float*) – The value of the function at the step length.

`TerminationCondition`(*f_tol: float*, *f_rtol: float*, *f0_norm: float*, *x_tol: float*, *x_rtol: float*)

Class to check the termination condition of the root finder.

class Jacobian

Base class for the Jacobians used in rootfinder algorithms.

A Jacobian can best be defined as a determinant which is defined for a finite number of functions of the same number of variables in which each row consists of the first partial derivatives of the same function with respect to each of the variables.

References

[1].. https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant [2].. Kasim, Muhammad & Vinko, Sam. (2020). xi$-torch: differentiable scientific computing library.

abstract setup(*x0: Tensor*, *y0: Tensor*, *func: Callable*)

Setup the Jacobian for the rootfinder.

abstract solve(*v: Tensor*, *tol: Any = 0*)

Solve the linear system *J dx = v*.

abstract update(*x: Tensor*, *y: Tensor*)

Update the Jacobian approximation.

class BroydenFirst(*alpha: Tensor | None = None*, *uv0: Any | None = None*, *max_rank: float | None = None*)

Approximating the Jacobian based on Broyden's first approximation.

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.jacobian import BroydenFirst
>>> jacobian = BroydenFirst()
>>> x0 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> def func(x):
...     return torch.tensor([x[0]**2 + x[1]**2 - 1.0, x[0] - x[1]])
>>> y0 = func(x0)
>>> v = torch.tensor([1.0, 1.0])
>>> jacobian.setup(x0, y0, func)
>>> jacobian.solve(v)
tensor([-0.7071, -0.7071], grad_fn=<MulBackward0>)
```

References

[1].. B.A. van der Rotten, PhD thesis,
"A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003).

__init__(*alpha: Tensor | None = None*, *uv0: Any | None = None*, *max_rank: float | None = None*)

The initial guess of inverse Jacobian is *-alpha * I + u v^T*. *max_rank* indicates the maximum rank of the Jacoabian before reducing it

Parameters

- **alpha** (*Union[torch.Tensor, None]*) – The initial guess of inverse Jacobian is *-alpha * I*. If None, it is set to *-1.0*.
- **uv0** (*tuple, optional*) – The initial guess of the inverse Jacobian.

- **max_rank** (*Union[float, None]*) – The maximum rank of the Jacobian before reducing it. If None, it is set to *inf*.

setup(*x0: Tensor*, *y0: Tensor*, *func: Callable*)

Setup the Jacobian for the rootfinder.

Parameters

- **x0** – The initial guess of the root.
- **y0** – The function value at the initial guess.
- **func** – The function to find the root.

solve(*v: Tensor*, *tol=0*) → Tensor

Solve the linear system $J\,dx = v$.

Parameters

- **v** (*torch.Tensor*) – The right-hand side of the linear system.
- **tol** (*torch.Tensor*) – The tolerance for the linear system.

Returns

res – The solution of the linear system.

Return type

torch.Tensor

update(*x: Tensor*, *y: Tensor*)

Update the Jacobian approximation.

Parameters

- **x** (*torch.Tensor*) – The current point.
- **y** (*torch.Tensor*) – The function value at the current point.

class BroydenSecond(*alpha: Tensor | None = None*, *uv0: Any | None = None*, *max_rank: float | None = None*)

Inverse Jacobian approximation based on Broyden's second method.

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.jacobian import BroydenSecond
>>> jacobian = BroydenSecond()
>>> x0 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> def func(x):
...     return torch.tensor([x[0]**2 + x[1]**2 - 1.0, x[0] - x[1]])
>>> y0 = func(x0)
>>> v = torch.tensor([1.0, 1.0])
>>> jacobian.setup(x0, y0, func)
>>> jacobian.solve(v)
tensor([-0.7071, -0.7071], grad_fn=<MulBackward0>)
```

References

[1] B.A. van der Rotten, PhD thesis,
"A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003).

class LinearMixing(*alpha: float | None = None*)

Approximating the Jacobian based on linear mixing. It acts as a simple check for the functionality of the rootfinder.

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.jacobian import LinearMixing
>>> jacobian = LinearMixing()
>>> x0 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> def func(x):
...     return torch.tensor([x[0]**2 + x[1]**2 - 1.0, x[0] - x[1]])
>>> y0 = func(x0)
>>> v = torch.tensor([1.0, 1.0])
>>> jacobian.setup(x0, y0, func)
>>> jacobian.solve(v)
tensor([1., 1.])
```

__init__(*alpha: float | None = None*)

The initial guess of inverse Jacobian is `-alpha * I`

Parameters
alpha (*float, optional*) – The initial guess of inverse Jacobian is `-alpha * I`. If None, it is set to `-1.0`.

setup(*x0: Tensor*, *y0: Tensor*, *func: Callable*)

Setup the Jacobian for the rootfinder.

Parameters

- **x0** (*torch.Tensor*) – The initial guess of the root.
- **y0** (*torch.Tensor*) – The function value at the initial guess.
- **func** (*Callable*) – The function to find the root.

solve(*v: Tensor*, *tol=0*) → Tensor

Solve the linear system $J\, dx = v$.

Parameters

- **v** (*torch.Tensor*) – The right-hand side of the linear system.
- **tol** – The tolerance for the linear system.

update(*x: Tensor*, *y: Tensor*)

Update the Jacobian approximation.

Parameters

- **x** (*torch.Tensor*) – The current point.
- **y** (*torch.Tensor*) – The function value at the current point.

class LowRankMatrix(*alpha: Tensor*, *uv0*, *reduce_method: str*)

represents a matrix of *lpha * I + sum_n c_n d_n^T*

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.jacobian import LowRankMatrix
>>> import torch
>>> alpha = 1.0
>>> uv0 = (torch.tensor([1.0, 1.0]), torch.tensor([1.0, 1.0]))
>>> reduce_method = "restart"
>>> matrix = LowRankMatrix(alpha, uv0, reduce_method)
>>> v = torch.tensor([1.0, 1.0])
>>> matrix.mv(v)
tensor([3., 3.])
>>> matrix.rmv(v)
tensor([3., 3.])
```

__init__(*alpha: Tensor*, *uv0*, *reduce_method: str*)

initialize the matrix

Parameters

- **alpha** (*torch.Tensor*) – The coefficient of the identity matrix
- **uv0** (*tuple*) – The initial guess of the inverse Jacobian
- **reduce_method** (*str*) – The method to reduce the rank of the matrix

mv(*v: Tensor*) → Tensor

multiply the matrix with a vector

Parameters

v (*torch.Tensor*) – Vector to multiply

Returns

res – Result of the multiplication

Return type

torch.Tensor

rmv(*v: Tensor*) → Tensor

multiply the transpose of the matrix with a vector

Parameters

v (*torch.Tensor*) – Vector to multiply

Returns

res – Result of the multiplication

Return type

torch.Tensor

append(*c: Tensor*, *d: Tensor*)

append a rank-1 matrix to the matrix

Parameters

- **c** (*torch.Tensor*) – The first vector
- **d** (*torch.Tensor*) – The second vector

Returns
res – The matrix after appending the rank-1 matrix

Return type
Union['LowRankMatrix', 'FullRankMatrix']

`reduce`(*max_rank: int*, ***otherparams*)
reduce the rank of the matrix

Parameters

- **max_rank** (*int*) – The maximum rank of the matrix
- **otherparams** – Other parameters

class `FullRankMatrix`(*alpha: Tensor*, *cns: Any*, *dns: Any*)
represents a full rank matrix of *lpha * I + sum_n c_n d_n^T*

Examples

```
>>> from deepchem.utils.differentiation_utils.optimize.jacobian import␣
↪FullRankMatrix
>>> import torch
>>> alpha = 1.0
>>> cns = [torch.tensor([1.0, 1.0]), torch.tensor([1.0, 1.0])]
>>> dns = [torch.tensor([1.0, 1.0]), torch.tensor([1.0, 1.0])]
>>> matrix = FullRankMatrix(alpha, cns, dns)
>>> v = torch.tensor([1.0, 1.0])
>>> matrix.mv(v)
tensor([5., 5.])
>>> matrix.rmv(v)
tensor([5., 5.])
```

`__init__`(*alpha: Tensor*, *cns: Any*, *dns: Any*)
initialize the matrix

Parameters

- **alpha** (*torch.Tensor*) – Coefficient of the identity matrix
- **cns** (*List*) – List of the first vectors
- **dns** (*List*) – List of the second vectors

`mv`(*v: Tensor*) → Tensor
multiply the matrix with a vector

Parameters
v (*torch.Tensor*) – The vector to multiply

Returns
res – The result of the multiplication

Return type
torch.Tensor

`rmv`(*v: Tensor*) → Tensor
multiply the transpose of the matrix with a vector

Parameters
`v` (*torch.Tensor*) – The vector to multiply

Returns
The result of the multiplication

Return type
torch.Tensor

`append`(*c: Tensor*, *d: Tensor*)

append a rank-1 matrix to the matrix

Parameters

- `c` (*torch.Tensor*) – The first vector
- `d` (*torch.Tensor*) – The second vector

Returns
The matrix after appending the rank-1 matrix

Return type
FullRankMatrix

`reduce`(*max_rank: int*, ***kwargs*)

reduce the rank of the matrix

Parameters

- `max_rank` (*int*) – The maximum rank of the matrix
- `otherparams` – Other parameters

`rootfinder`(*fcn: Callable[[...], Tensor]*, *y0: Tensor*, *params: Sequence[Any] = []*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tensor

Solving the rootfinder equation of a given function,

$$\mathbf{0} = \mathbf{f}(\mathbf{y}, \theta)$$

where $\mathbf{f}$ is a function that can be non-linear and produce output of the same shape of $\mathbf{y}$, and θ is other parameters required in the function. The output of this block is $\mathbf{y}$ that produces the $\mathbf{0}$ as the output.

Parameters

- `fcn` (*callable*) – The function $\mathbf{f}$ with output tensor `(*ny)`
- `y0` (*torch.tensor*) – Initial guess of the solution with shape `(*ny)`
- `params` (*list*) – Sequence of any other parameters to be put in `fcn`
- `bck_options` (*dict*) – Method-specific options for the backward solve (see `xitorch.linalg.solve()`)
- `method` (*str or callable or None*) – Rootfinder method. If None, it will choose `"broyden1"`.
- `**fwd_options` – Method-specific options (see method section)

Returns
The solution which satisfies $\mathbf{0} = \mathbf{f}(\mathbf{y}, \theta)$ with shape `(*ny)`

Return type
torch.tensor

Example

```
>>> import torch
>>> def func1(y, A):  # example function
...     return torch.tanh(A @ y + 0.1) + y / 2.0
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> yroot = rootfinder(func1, y0, params=(A,))
>>> print(yroot)
tensor([[-0.0459],
        [-0.0663]], grad_fn=<_RootFinderBackward>)
```

equilibrium(*fcn: Callable[[...], Tensor]*, *y0: Tensor*, *params: Sequence[Any] = []*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tensor

Solving the equilibrium equation of a given function,

$$\mathbf{y} = \mathbf{f}(\mathbf{y}, \theta)$$

where $\mathbf{f}$ is a function that can be non-linear and produce output of the same shape of $\mathbf{y}$, and θ is other parameters required in the function. The output of this block is $\mathbf{y}$ that produces the same $\mathbf{y}$ as the output.

Parameters

- **fcn** (*callable*) – The function $\mathbf{f}$ with output tensor `(*ny)`
- **y0** (*torch.tensor*) – Initial guess of the solution with shape `(*ny)`
- **params** (*list*) – Sequence of any other parameters to be put in `fcn`
- **bck_options** (*dict*) – Method-specific options for the backward solve (see `xitorch.linalg.solve()`)
- **method** (*str or None*) – Rootfinder method. If None, it will choose `"broyden1"`.
- ****fwd_options** – Method-specific options (see method section)

Returns
The solution which satisfies $\mathbf{y} = \mathbf{f}(\mathbf{y}, \theta)$ with shape `(*ny)`

Return type
torch.tensor

Example

```
>>> import torch
>>> def func1(y, A):  # example function
...     return torch.tanh(A @ y + 0.1) + y / 2.0
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> yequil = equilibrium(func1, y0, params=(A,))
>>> print(yequil)
tensor([[ 0.2313],
        [-0.5957]], grad_fn=<_RootFinderBackward>)
```

Note:

- This is a direct implementation of finding the root of $\mathbf{g}(\mathbf{y}, \theta) = \mathbf{y} - \mathbf{f}(\mathbf{y}, \theta)$

minimize(*fcn: Callable[[...], Tensor]*, *y0: Tensor*, *params: Sequence[Any] = []*, *bck_options: Mapping[str, Any] = {}*, *method: str | Callable | None = None*, ***fwd_options*) → Tensor

Solve the unbounded minimization problem:

$$\mathbf{y}^* = \arg\min_{\mathbf{y}} f(\mathbf{y}, \theta)$$

to find the best $\mathbf{y}$ that minimizes the output of the function f.

Parameters

- **fcn** (*callable*) – The function to be optimized with output tensor with 1 element.
- **y0** (*torch.tensor*) – Initial guess of the solution with shape `(*ny)`
- **params** (*list*) – Sequence of any other parameters to be put in `fcn`
- **bck_options** (*dict*) – Method-specific options for the backward solve (see `xitorch.linalg.solve()`)
- **method** (*str or callable or None*) – Minimization method. If None, it will choose `"broyden1"`.
- ****fwd_options** – Method-specific options (see method section)

Returns
The solution of the minimization with shape `(*ny)`

Return type
torch.tensor

Example

```
>>> import torch
>>> def func1(y, A):  # example function
...     return torch.sum((A @ y)**2 + y / 2.0)
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> ymin = minimize(func1, y0, params=(A,))
>>> print(ymin)
tensor([[-0.0519],
        [-0.2684]], grad_fn=<_RootFinderBackward>)
```

class _RootFinder(**args*, ***kwargs*)

static forward(*ctx*, *fcn*, *y0*, *fwd_fcn*, *alg_type*, *options*, *bck_options*, *nparams*, **allparams*)

Forward method for the rootfinder, minimizer, and equilibrium

Parameters

- **fcn** – a function that returns what has to be 0 (will be used in the backward, not used in the forward). For minimization, it is the gradient
- **y0** – initial guess
- **fwd_fcn** – a function that will be executed in the forward method (unused in the backward)
- **alg_type** – the type of algorithm: “rootfinder”, “minimizer”, or “equilibrium”
- **options** – options for the forward method

- **bck_options** – options for the backward method
- **nparams** – number of parameters
- **allparams** – all parameters (including the non-tensor parameters)
- **minimization** (*This class is also used for*) –
- **might** (*where fcn and fwd_fcn*) –
- **different** (*be slightly*) –

Returns
The solution of the rootfinder, minimizer, or equilibrium

Return type
torch.tensor

static backward(*ctx*, *grad_yout*)

Backward method for the rootfinder, minimizer, and equilibrium

Parameters
grad_yout (*torch.tensor*) – the gradient of the output of the rootfinder, minimizer, or equilibrium

Returns
The gradients of the parameters

Return type
tuple

_get_rootfinder_default_method(*method: str | Callable | None = None*) → str | Callable | None

Get the default method for the rootfinder, minimizer, and equilibrium

Examples

```
>>> _get_rootfinder_default_method(None)
'broyden1'
```

Parameters
method (*str or None*) – The method name

Returns
The method name

Return type
str

_get_equilibrium_default_method(*method: str | Callable | None = None*) → str | Callable | None

Get the default method for the equilibrium

Examples

```
>>> _get_equilibrium_default_method(None)
'broyden1'
```

Parameters
method (*str or None*) – The method name

Returns
The method name

Return type
str

_get_minimizer_default_method(*method: str | Callable | None = None*) → str | Callable | None

Get the default method for the minimizer

Examples

```
>>> _get_minimizer_default_method(None)
'broyden1'
```

Parameters
method (*str or None*) – The method name

Returns
The method name

Return type
str

3.31.18 Attribute Utilities

The utilities here are used to modify the attributes of the classes. Used by differentiation_utils.

class **get_attr**(*obj: object*, *name: str*)

Get the attribute of an object.

Examples

```
>>> from deepchem.utils.attribute_utils import get_attr
>>> class MyClass:
...     def __init__(self):
...         self.a = 1
...         self.b = 2
>>> obj = MyClass()
>>> get_attr(obj, "a")
1
```

Parameters

- **obj** (*object*) – The object to get the attribute from.

- **name** (*str*) – The name of the attribute.

Returns
val – The value of the attribute.

Return type
object

class set_attr(*obj: object*, *name: str*, *val: object*)

Set the attribute of an object.

Examples

```
>>> from deepchem.utils import set_attr
>>> class MyClass:
...     def __init__(self):
...         self.a = 1
...         self.b = 2
>>> obj = MyClass()
>>> set_attr(obj, "a", 3)
>>> set_attr(obj, "c", 4)
>>> obj.a
3
>>> obj.c
4
```

Parameters

- **obj** (*object*) – The object to set the attribute to.
- **name** (*str*) – The name of the attribute.
- **val** (*object*) – The value to set the attribute to.

class del_attr(*obj: Any*, *name: str*)

Delete the attribute of an object.

Examples

```
>>> from deepchem.utils import del_attr
>>> class MyClass:
...     def __init__(self):
...         self.a = 1
...         self.b = 2
>>> obj = MyClass()
>>> del_attr(obj, "a")
>>> try:
...     obj.a
... except AttributeError:
...     print("AttributeError")
AttributeError
```

3.31.19 Pytorch Utilities

unsorted_segment_sum(*data: Tensor*, *segment_ids: Tensor*, *num_segments: int*) → Tensor

Computes the sum along segments of a tensor. Analogous to tf.unsorted_segment_sum.

Parameters

- **data** (*torch.Tensor*) – A tensor whose segments are to be summed.
- **segment_ids** (*torch.Tensor*) – The segment indices tensor.
- **num_segments** (*int*) – The number of segments.

Returns
tensor

Return type
torch.Tensor

Examples

```
>>> segment_ids = torch.Tensor([0, 1, 0]).to(torch.int64)
>>> data = torch.Tensor([[1, 2, 3, 4], [5, 6, 7, 8], [4, 3, 2, 1]])
>>> num_segments = 2
>>> result = unsorted_segment_sum(data=data,
...                               segment_ids=segment_ids,
...                               num_segments=num_segments)
>>> data.shape[0]
3
>>> segment_ids.shape[0]
3
>>> len(segment_ids.shape)
1
>>> result
tensor([[5., 5., 5., 5.],
        [5., 6., 7., 8.]])
```

segment_sum(*data: Tensor*, *segment_ids: Tensor*) → Tensor

This function computes the sum of values along segments within a tensor. It is useful when you have a tensor with segment IDs and you want to compute the sum of values for each segment. This function is analogous to tf.segment_sum. (https://www.tensorflow.org/api_docs/python/tf/math/segment_sum).

Parameters

- **data** (*torch.Tensor*) – A pytorch tensor containing the values to be summed. It can have any shape, but its rank (number of dimensions) should be at least 1.
- **segment_ids** (*torch.Tensor*) – A 1-D tensor containing the indices for the segmentation. The segments can be any non-negative integer values, but they must be sorted in non-decreasing order.

Returns
out_tensor – Tensor with the same shape as data, where each value corresponds to the sum of values within the corresponding segment.

Return type
torch.Tensor

Examples

```
>>> data = torch.Tensor([[1, 2, 3, 4], [4, 3, 2, 1], [5, 6, 7, 8]])
>>> segment_ids = torch.Tensor([0, 0, 1]).to(torch.int64)
>>> result = segment_sum(data=data, segment_ids=segment_ids)
>>> data.shape[0]
3
>>> segment_ids.shape[0]
3
>>> len(segment_ids.shape)
1
>>> result
tensor([[5., 5., 5., 5.],
        [5., 6., 7., 8.]])
```

chunkify(*a: Tensor*, *dim: int*, *maxnumel: int*) → Generator[Tuple[Tensor, int, int], None, None]

Splits the tensor *a* into several chunks of size *maxnumel* along the dimension given by *dim*.

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import chunkify
>>> a = torch.arange(10)
>>> for chunk, istart, iend in chunkify(a, 0, 3):
...     print(chunk, istart, iend)
tensor([0, 1, 2]) 0 3
tensor([3, 4, 5]) 3 6
tensor([6, 7, 8]) 6 9
tensor([9]) 9 12
```

Parameters

- **a** (*torch.Tensor*) – The big tensor to be splitted into chunks.
- **dim** (*int*) – The dimension where the tensor would be splitted.
- **maxnumel** (*int*) – Maximum number of elements in a chunk.

Returns

chunks – A generator that yields a tuple of three elements: the chunk tensor, the starting index of the chunk and the ending index of the chunk.

Return type

Generator[Tuple[torch.Tensor, int, int], None, None]

get_memory(*a: Tensor*) → int

Returns the size of the tensor in bytes.

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import get_memory
>>> a = torch.randn(100, 100, dtype=torch.float64)
>>> get_memory(a)
80000
```

Parameters
a (*torch.Tensor*) – The tensor to be measured.

Returns
size – The size of the tensor in bytes.

Return type
int

gaussian_integral(*n: int*, *alpha: float | Tensor*) → float | Tensor

Performs the gaussian integration.

Examples

```
>>> gaussian_integral(5, 1.0)
1.0
```

Parameters

- **n** (*int*) – The order of the integral
- **alpha** (*Union[float, torch.Tensor]*) – The parameter of the gaussian

Returns
The value of the integral

Return type
Union[float, torch.Tensor]

TensorNonTensorSeparator(*params: Sequence*, *varonly: bool = True*)

Class that provides function to separate/combine tensors and nontensors parameters.

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import TensorNonTensorSeparator
>>> a = torch.tensor([1.,2,3])
>>> b = 4.
>>> c = torch.tensor([5.,6,7], requires_grad=True)
>>> params = [a, b, c]
>>> separator = TensorNonTensorSeparator(params)
>>> tensor_params = separator.get_tensor_params()
>>> tensor_params
[tensor([5., 6., 7.], requires_grad=True)]
```

`tallqr`(*V*, *MV=None*)

QR decomposition for tall and skinny matrix.

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import tallqr
>>> V = torch.randn(3, 2)
>>> Q, R = tallqr(V)
>>> Q.shape
torch.Size([3, 2])
>>> R.shape
torch.Size([2, 2])
>>> torch.allclose(Q @ R, V)
True
```

Parameters

- **V** (*torch.Tensor*) – V is a matrix to be decomposed. (**BV*, na, nguess)
- **MV** (*torch.Tensor*) – (**BM*, na, nguess) where M is the basis to make Q M-orthogonal if MV is None, then MV=V (default=None)

Returns

- **Q** (*torch.Tensor*) – The Orthogonal Part. Shape: (**BV*, na, nguess)
- **R** (*torch.Tensor*) – The (**BM*, nguess, nguess) where M is the basis to make Q M-orthogonal

`to_fortran_order`(*V*)

Convert a tensor to Fortran order. (The last two dimensions are made Fortran order.) Fortran order/ array is a special case in which all elements of an array are stored in column-major order.

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import to_fortran_order
>>> V = torch.randn(3, 2)
>>> V.is_contiguous()
True
>>> V = to_fortran_order(V)
>>> V.is_contiguous()
False
>>> V.shape
torch.Size([3, 2])
>>> V = torch.randn(3, 2).transpose(-2, -1)
>>> V.is_contiguous()
False
>>> V = to_fortran_order(V)
>>> V.is_contiguous()
False
>>> V.shape
torch.Size([2, 3])
```

Parameters
`V` (*torch.Tensor*) – V is a matrix to be converted. (**BV*, na, nguess)

Returns
outV – (**BV*, nguess, na)

Return type
torch.Tensor

`get_np_dtype`(*dtype: dtype*) → Any
corresponding numpy dtype from the input pytorch's tensor dtype

Examples

```
>>> import torch
>>> from deepchem.utils.pytorch_utils import get_np_dtype
>>> get_np_dtype(torch.float32)
<class 'numpy.float32'>
>>> get_np_dtype(torch.float64)
<class 'numpy.float64'>
```

Parameters
`dtype` (*torch.dtype*) – pytorch's tensor dtype

Returns
corresponding numpy dtype

Return type
np.dtype

3.31.20 Batch Utilities

The utilites here are used for computing features on batch of data. Can be used inside of default_generator function.

`batch_coulomb_matrix_features`(*X_b: ndarray*, *distance_max: float = -1*, *distance_min: float = 18*, *n_distance: int = 100*)

Computes the values for different Feature on given batch. It works as a helper function to coulomb matrix.

This function takes in a batch of Molecules represented as Coulomb Matrix.

It proceeds as follows:

- It calculates the Number of atoms per molecule by counting all the non zero elements(numbers) of every molecule layer in matrix in one dimension.
- The Gaussian distance is calculated using the Euclidean distance between the Cartesian coordinates of two atoms. The distance value is then passed through a Gaussian function, which transforms it into a continuous value.
- Then using number of atom per molecule, calculates the atomic charge by looping over the molecule layer in the Coulomb matrix and takes the *2.4* root of the diagonal of *2X* of each molecule layer. *Undoing the Equation of coulomb matrix.*
- Atom_membership is assigned as a commomn repeating integers for all the atoms for a specific molecule.
- Distance Membership encodes spatial information, assigning closer values to atoms that are in that specific molecule. All initial Distances are added a start value to them which are unique to each molecule.

Models Used in:

- DTNN

Parameters

- **X_b** (*np.ndarray*) – It is a 3d Matrix containing information of each the atom's ionic interaction with other atoms in the molecule.
- **distance_min** (*float (default -1)*) – minimum distance of atom pairs (in Angstrom)
- **distance_max** (*float (default = 18)*) – maximum distance of atom pairs (in Angstrom)
- **n_distance** (*int (default 100)*) – granularity of distance matrix step size will be (distance_max-distance_min)/n_distance

Returns

- **atom_number** (*np.ndarray*) – Atom numbers are assigned to each atom based on their atomic properties. The atomic numbers are derived from the periodic table of elements. For example, hydrogen -> 1, carbon -> 6, and oxygen -> 8.
- **gaussian_dist** (*np.ndarray*) – Gaussian distance refers to the method of representing the pairwise distances between atoms in a molecule using Gaussian functions. The Gaussian distance is calculated using the Euclidean distance between the Cartesian coordinates of two atoms. The distance value is then passed through a Gaussian function, which transforms it into a continuous value.
- **atom_mem** (*np.ndarray*) – Atom membership refers to the binary representation of whether an atom belongs to a specific group or property within a molecule. It allows the model to incorporate domain-specific information and enhance its understanding of the molecule's properties and interactions.
- **dist_mem_i** (*np.ndarray*) – Distance membership i are utilized to encode spatial information and capture the influence of atom distances on the properties and interactions within a molecule. The inner membership function assigns higher values to atoms that are closer to the atoms' interaction region, thereby emphasizing the impact of nearby atoms.
- **dist_mem_j** (*np.ndarray*) – It captures the long-range effects and influences between atoms that are not in direct proximity but still contribute to the overall molecular properties. Distance membership j are utilized to encode spatial information and capture the influence of atom distances on the properties and interactions outside a molecule. The outer membership function assigns higher values to atoms that are farther to the atoms' interaction region, thereby emphasizing the impact of farther atoms.

Examples

```
>>> import os
>>> import deepchem as dc
>>> current_dir = os.path.dirname(os.path.abspath(__file__))
>>> dataset_file = os.path.join(current_dir, 'test/assets/qm9_mini.sdf')
>>> TASKS = ["alpha", "homo"]
>>> loader = dc.data.SDFLoader(tasks=TASKS,
...                            featurizer=dc.feat.CoulombMatrix(29),
...                            sanitize=True)
>>> data = loader.create_dataset(dataset_file, shard_size=100)
>>> inputs = dc.utils.batch_utils.batch_coulomb_matrix_features(data.X)
```

References

batch_elements(*elements: List[Any]*, *batch_size: int*)

Combine elements into batches.

Parameters

- **elements** (*List[Any]*) – List of Elements to be combined into batches.
- **batch_size** (*int*) – Batch size in which to divide.

Returns
batch – List of Lists of elements divided into batches.

Return type
List[Any]

Examples

```
>>> import deepchem as dc
>>> # Prepare Data
>>> inputs = [[i, i**2, i**3] for i in range(10)]
>>> # Run
>>> output = list(dc.utils.batch_utils.batch_elements(inputs, 3))
>>> len(output)
4
```

create_input_array(*sequences: Collection*, *max_input_length: int*, *reverse_input: bool*, *batch_size: int*, *input_dict: Dict*, *end_mark: Any*)

Create the array describing the input sequences.

It creates a 2d Matrix empty matrix according to batch size and max_length. Then iteratively fills it with the key-values from the input dictionary.

Many NLP Models like SeqToSeq has sentences as there inputs. We need to convert these sentences into numbers so that the model can do computation on them.

This function takes in the sentence then using the *input_dict* dictionary picks up the words/letters equivalent numerical represntation. Then makes an numpy array of it.

If the *reverse_input* is True, then the order of the input sequences is reversed before sending them into the encoder. This can improve performance when working with long sequences.

These values can be used to generate embeddings for further processing.

Models used in:

- SeqToSeq

Parameters

- **sequences** (*Collection*) – List of sequences to be converted into input array.
- **reverse_input** (*bool*) – If True, reverse the order of input sequences before sending them into the encoder. This can improve performance when working with long sequences.
- **batch_size** (*int*) – Batch size of the input array.
- **input_dict** (*dict*) – Dictionary containing the key-value pairs of input sequences.
- **end_mark** (*Any*) – End mark for the input sequences.

Returns
features – Numeric Representation of the given sequence according to input_dict.

Return type
np.Array

Examples

```
>>> import deepchem as dc
>>> # Prepare Data
>>> inputs = [["a", "b"], ["b", "b", "b"]]
>>> input_dict = {"c": 0, "a": 1, "b": 2}
>>> # Inputs property
>>> max_length = max([len(x) for x in inputs])
>>> # Without reverse input
>>> output_1 = dc.utils.batch_utils.create_input_array(inputs, max_length,
...                                                    False, 2, input_dict,
...                                                    "c")
>>> output_1.shape
(2, 4)
>>> # With revercse input
>>> output_2 = dc.utils.batch_utils.create_input_array(inputs, max_length,
...                                                    True, 2, input_dict,
...                                                    "c")
>>> output_2.shape
(2, 4)
```

`create_output_array`(*sequences: Collection*, *max_output_length: int*, *batch_size: int*, *output_dict: Dict*, *end_mark: Any*)

Create the array describing the target sequences.

It creates a 2d Matrix empty matrix according to batch size and max_length. Then iteratively fills it with the key-values from the output dictionary.

This function is similar to *create_input_array* function. The only difference is that it is used for output sequences and does not have the *reverse_input* parameter as it is not required for output sequences.

It is used in NLP Models like SeqToSeq where the output is also a sentence and we need to convert it into numbers so that the model can do computation on them. This function takes in the sentence then using the *output_dict* dictionary picks up the words/letters equivalent numerical represntation. Then makes an numpy array of it.

These values can be used to generate embeddings for further processing.

Models used in:

- SeqToSeq

Parameters

- **sequences** (*Collection*) – List of sequences to be converted into output array.
- **max_output_length** (*bool*) – Maximum length of output sequence that may be generated
- **batch_size** (*int*) – Batch size of the output array.
- **output_dict** (*dict*) – Dictionary containing the key-value pairs of output sequences.
- **end_mark** (*Any*) – End mark for the output sequences.

Returns
features – Numeric Representation of the given sequence according to output_dict.

Return type
np.Array

Examples

```
>>> import deepchem as dc
>>> # Prepare Data
>>> inputs = [["a", "b"], ["b", "b", "b"]]
>>> output_dict = {"c": 0, "a": 1, "b": 2}
>>> # Inputs property
>>> max_length = max([len(x) for x in inputs])
>>> output = dc.utils.batch_utils.create_output_array(inputs, max_length, 2,
...                                                   output_dict, "c")
>>> output.shape
(2, 3)
```

3.31.21 Periodic Table Utilities

The Utilities here are used to computing atomic mass and radii data. These can be used by DFT and many other Molecular Models.

`get_atomz`(*element: str* | *int* | *float* | *Tensor*) → int | float | Tensor

Returns the atomic number for the given element

Examples

```
>>> from deepchem.utils import get_atomz
>>> element_symbol = "Al" # Aluminium
>>> get_atomz(element_symbol)
13
>>> get_atomz(17)
17
```

Parameters
element (`Union[str, ZType]`) – String symbol of Element or Atomic Number. Ex: H, He, C

Returns
atom_n – Atomic Number of the given Element.

Return type
ZType

References

Kasim, Muhammad F., and Sam M. Vinko. "Learning the exchange-correlation functional from nature with fully differentiable density functional theory." Physical Review Letters 127.12 (2021): 126403. https://github.com/diffqc/dqc/blob/master/dqc/utils/periodictable.py

3.31.22 Equivariance Utilities

The utilities here refer to equivariance tools that play a vital role in mathematics and applied sciences. They excel in preserving the relationships between objects or data points when undergoing transformations such as rotations or scaling.

You can refer to the tutorials for additional information regarding equivariance and Deepchem's support for equivariance.

su2_generators(*k: int*) → Tensor

Generate the generators of the special unitary group SU(2) in a given representation.

The function computes the generators of the SU(2) group for a specific representation determined by the value of 'k'. These generators are commonly used in the study of quantum mechanics, angular momentum, and related areas of physics and mathematics. The generators are represented as matrices.

The SU(2) group is a fundamental concept in quantum mechanics and symmetry theory. The generators of the group, denoted as J_x, J_y, and J_z, represent the three components of angular momentum operators. These generators play a key role in describing the transformation properties of physical systems under rotations.

The returned tensor contains three matrices corresponding to the x, y, and z generators, usually denoted as J_x, J_y, and J_z. These matrices form a basis for the Lie algebra of the SU(2) group.

In linear algebra, specifically within the context of quantum mechanics, lowering and raising operators are fundamental concepts that play a crucial role in altering the eigenvalues of certain operators while acting on quantum states. These operators are often referred to collectively as "ladder operators."

A lowering operator is an operator that, when applied to a quantum state, reduces the eigenvalue associated with a particular observable. In the context of SU(2), the lowering operator corresponds to **J_**-.

Conversely, a raising operator is an operator that increases the eigenvalue of an observable when applied to a quantum state. In the context of SU(2), the raising operator corresponds to J_+.

The z-generator matrix represents the component of angular momentum along the z-axis, often denoted as J_z. It commutes with both J_x and J_y and is responsible for quantizing the angular momentum.

Note that the dimensions of the returned tensor will be (3, 2j+1, 2j+1), where each matrix has a size of (2j+1) x (2j+1). :param k: The representation index, which determines the order of the representation. :type k: int

Returns
A stack of three SU(2) generators, corresponding to J_x, J_z, and J_y.

Return type
torch.Tensor

Notes

A generating set of a group is a subset S of the group G such that every element of G can be expressed as a combination (under the group operation) of finitely many elements of the subset S and their inverses.

The special unitary group $SU_n(q)$ is the set of $n*n$ unitary matrices with determinant +1. $SU(2)$ is homeomorphic with the orthogonal group $O_3^+(2)$. It is also called the unitary unimodular group and is a Lie group.

References

Examples

```
>>> su2_generators(1)
tensor([[[ 0.0000+0.0000j,  0.7071+0.0000j,  0.0000+0.0000j],
         [-0.7071+0.0000j,  0.0000+0.0000j,  0.7071+0.0000j],
         [ 0.0000+0.0000j, -0.7071+0.0000j,  0.0000+0.0000j]],

        [[-0.0000-1.0000j,  0.0000+0.0000j,  0.0000+0.0000j],
         [ 0.0000+0.0000j,  0.0000+0.0000j,  0.0000+0.0000j],
         [ 0.0000+0.0000j,  0.0000+0.0000j,  0.0000+1.0000j]],

        [[ 0.0000-0.0000j,  0.0000+0.7071j,  0.0000-0.0000j],
         [ 0.0000+0.7071j,  0.0000-0.0000j,  0.0000+0.7071j],
         [ 0.0000-0.0000j,  0.0000+0.7071j,  0.0000-0.0000j]]])
```

so3_generators(*k: int*) → Tensor

Construct the generators of the SO(3) Lie algebra for a given quantum angular momentum.

The function generates the generators of the special orthogonal group SO(3), which represents the group of rotations in three-dimensional space. Its Lie algebra, which consists of the generators of infinitesimal rotations, is often used in physics to describe angular momentum operators. The generators of the Lie algebra can be related to the SU(2) group, and this function uses a transformation to convert the SU(2) generators to the SO(3) basis.

The primary significance of the SO(3) group lies in its representation of three-dimensional rotations. Each matrix in SO(3) corresponds to a unique rotation, capturing the intricate ways in which objects can be oriented in 3D space. This concept finds application in numerous fields, ranging from physics to engineering.

Parameters
k (*int*) – The representation index, which determines the order of the representation.

Returns
A stack of three SO(3) generators, corresponding to J_x, J_z, and J_y.

Return type
torch.Tensor

Notes

The special orthogonal group $SO_n(q)$ is the subgroup of the elements of general orthogonal group $GO_n(q)$ with determinant 1. SO_3 (often written $SO(3)$) is the rotation group for three-dimensional space.

These matrices are orthogonal, which means their rows and columns form mutually perpendicular unit vectors. This preservation of angles and lengths makes orthogonal matrices fundamental in various mathematical and practical applications.

The "special" part of $SO(3)$ refers to the determinant of these matrices being $+1$. The determinant is a scalar value that indicates how much a matrix scales volumes. A determinant of $+1$ ensures that the matrix represents a rotation in three-dimensional space without involving any reflection or scaling operations that would reverse the orientation of space.

References

Examples

```
>>> so3_generators(1)
tensor([[[ 0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000, -1.0000],
         [ 0.0000,  1.0000,  0.0000]],

        [[ 0.0000,  0.0000,  1.0000],
         [ 0.0000,  0.0000,  0.0000],
         [-1.0000,  0.0000,  0.0000]],

        [[ 0.0000, -1.0000,  0.0000],
         [ 1.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000]]])
```

change_basis_real_to_complex(*k: int*, *dtype: dtype | None = None*, *device: device | None = None*) → Tensor

Construct a transformation matrix to change the basis from real to complex spherical harmonics.

This function constructs a transformation matrix Q that converts real spherical harmonics into complex spherical harmonics. It operates on the basis functions $Y_{ell m}$ and Y_{ell}^{m}, and accounts for the relationship between the real and complex forms of these harmonics as defined in the provided mathematical expressions.

The resulting transformation matrix Q is used to change the basis of vectors or tensors of real spherical harmonics to their complex counterparts.

Parameters

- **k** (*int*) – The representation index, which determines the order of the representation.
- **dtype** (*torch.dtype, optional*) – The data type for the output tensor. If not provided, the function will infer it. Default is None.
- **device** (*torch.device, optional*) – The device where the output tensor will be placed. If not provided, the function will use the default device. Default is None.

Returns

A transformation matrix Q that changes the basis from real to complex spherical harmonics.

Return type

torch.Tensor

Notes

Spherical harmonics Y_l^m are a family of functions that are defined on the surface of a unit sphere. They are used to represent various physical and mathematical phenomena that exhibit spherical symmetry. The indices l and m represent the degree and order of the spherical harmonics, respectively.

The conversion from real to complex spherical harmonics is achieved by applying specific transformation coefficients to the real-valued harmonics. These coefficients are derived from the properties of spherical harmonics.

References

Examples

# The transformation matrix generated is used to change the basis of a vector of # real spherical harmonics with representation index 1 to complex spherical harmonics. >>> change_basis_real_to_complex(1) tensor([[-0.7071+0.0000j, 0.0000+0.0000j, 0.0000-0.7071j],

> [0.0000+0.0000j, 0.0000-1.0000j, 0.0000+0.0000j], [-0.7071+0.0000j, 0.0000+0.0000j, 0.0000+0.7071j]])

wigner_D(*k: int*, *alpha: Tensor*, *beta: Tensor*, *gamma: Tensor*) → Tensor

Wigner D matrix representation of the SO(3) rotation group.

The function computes the Wigner D matrix representation of the SO(3) rotation group for a given representation index 'k' and rotation angles 'alpha', 'beta', and 'gamma'. The resulting matrix satisfies properties of the SO(3) group representation.

Parameters

- **k** (*int*) – The representation index, which determines the order of the representation.
- **alpha** (*torch.Tensor*) – Rotation angles (in radians) around the Y axis, applied third.
- **beta** (*torch.Tensor*) – Rotation angles (in radians) around the X axis, applied second.
- **gamma** (*torch.Tensor)*) – Rotation angles (in radians) around the Y axis, applied first.

Returns

The Wigner D matrix of shape (#angles, 2k+1, 2k+1).

Return type

torch.Tensor

Notes

The Wigner D-matrix is a unitary matrix in an irreducible representation of the groups SU(2) and SO(3).

The Wigner D-matrix is used in quantum mechanics to describe the action of rotations on states of particles with angular momentum. It is a key concept in the representation theory of the rotation group SO(3), and it plays a crucial role in various physical contexts.

Examples

```
>>> k = 1
>>> alpha = torch.tensor([0.1, 0.2])
>>> beta = torch.tensor([0.3, 0.4])
>>> gamma = torch.tensor([0.5, 0.6])
>>> wigner_D_matrix = wigner_D(k, alpha, beta, gamma)
>>> wigner_D_matrix
tensor([[[ 0.8275,  0.1417,  0.5433],
         [ 0.0295,  0.9553, -0.2940],
         [-0.5607,  0.2593,  0.7863]],

        [[ 0.7056,  0.2199,  0.6737],
         [ 0.0774,  0.9211, -0.3816],
         [-0.7044,  0.3214,  0.6329]]])
```

3.31.23 Miscellaneous Utilities

The utilities here are used for miscellaneous purposes. Initial usecases are for improving the printing format of __repr__.

indent(*s*, *nspace*)

Gives indentation of the second line and next lines. It is used to format the string representation of an object. Which might be containing multiples objects in it. Usage: LinearOperator

Parameters

- **s** (*str*) – The string to be indented.
- **nspace** (*int*) – The number of spaces to be indented.

Returns
The indented string.

Return type
str

shape2str(*shape*)

Convert the shape to string representation. It also nicely formats the shape to be readable.

Parameters
shape (*Sequence[int]*) – The shape to be converted to string representation.

Returns
The string representation of the shape.

Return type
str

class UnimplementedError

Raised if a method is not implemented.

class GetSetParamsError

Raised if there is an error in getting or setting parameters.

class ConvergenceWarning

Warning to be raised if the convergence of an algorithm is not achieved.

`class MathWarning`

Raised if there are mathematical conditions that are not satisfied.

`class Uniquifier`(*allobjs: List*)

Identifies and tracks unique objects within a list, even if they are duplicates based on internal memory addresses (using id()). It Optimizes operations involving unique objects by avoiding redundant processing.

Examples

```
>>> from deepchem.utils import Uniquifier
>>> a = 1
>>> b = 2
>>> c = 3
>>> d = 1
>>> u = Uniquifier([a, b, c, a, d])
>>> u.get_unique_objs()
[1, 2, 3]
```

`__init__`(*allobjs: List*)

Initialize the uniquifier.

Parameters
`allobjs` (*List*) – The list of objects to be uniquified.

`get_unique_objs`(*allobjs: List | None = None*) → List

Get the unique objects.

Parameters
`allobjs` (*Optional[List]*) – The list of objects to be uniquified.

Returns
The list of unique objects.

Return type
List

`map_unique_objs`(*uniqueobjs: List*) → List

Map the unique objects to the original objects.

Parameters
`uniqueobjs` (*List*) – The list of unique objects.

3.31.24 SafeOperations Utilities

The utilities here are used for safe operations on tensors. These are used to avoid NaNs and Infs in the output.

`safepow`(*a: Tensor*, *p: Tensor*, *eps: float = 1e-12*) → Tensor

Safely calculate the power of a tensor with a small eps to avoid nan.

Examples

```
>>> import torch
>>> a = torch.tensor([1e-35, 2e-40])
>>> p = torch.tensor([2., 3])
>>> safepow(a, p)
tensor([1.0000e-24, 1.0000e-36])
>>> a**p
tensor([0., 0.])
```

Parameters

- **a** (*torch.Tensor*) – Base tensor on which to calculate the power. Must be positive.
- **p** (*torch.Tensor*) – Power tensor, by which to calculate the power.
- **eps** (*float (default 1e-12)*) – The eps to add to the base tensor.

Returns
The result tensor.

Return type
torch.Tensor

Raises
RuntimeError – If the base tensor contains negative values.

safenorm(*a: Tensor*, *dim: int*, *eps: float = 1e-15*) → Tensor

Calculate the 2-norm safely. The square root of the inner product of a vector with itself.

Examples

```
>>> import torch
>>> a = torch.tensor([1e-35, 2e-40])
>>> safenorm(a, 0)
tensor(1.4142e-15)
>>> a.norm()
tensor(0.)
```

Parameters

- **a** (*torch.Tensor*) – The tensor to calculate the norm.
- **dim** (*int*) – The dimension to calculate the norm.
- **eps** (*float (default 1e-15)*) – The eps to add to the base tensor.

Returns
The result tensor.

Return type
torch.Tensor

occnumber(*a: int | float | Tensor*, *n: int | None = None*, *dtype: dtype = torch.float64*, *device: device = device(type='cpu')*) → Tensor

Occupation number (maxed at 1) where the total sum of the output equals to a with length of the output is n.

Examples

```
>>> import torch
>>> occnumber(torch.tensor(2.5), 3, torch.double, torch.device('cpu'))
tensor([1.0000, 1.0000, 0.5000], dtype=torch.float64)
>>> occnumber(2.5)
tensor([1.0000, 1.0000, 0.5000], dtype=torch.float64)
```

Parameters

- **a** (*ZType*) – Total sum of the output
- **n** (*Optional[int] (default None)*) – Length of the output
- **dtype** (*torch.dtype (default torch.double)*) – Data type of the output
- **device** (*torch.device (default torch.device('cpu'))*) – Device of the output

Returns
The constructed occupation number

Return type
torch.Tensor

get_floor_and_ceil(*aa: int | float*) → Tuple[int, int]

get the ceiling and flooring of aa.

Examples

```
>>> get_floor_and_ceil(2.5)
(2, 3)
```

Parameters
aa (*Union[int, float]*) – The input number

Returns
The flooring and ceiling of aa

Return type
Tuple[int, int]

safe_cdist(*a: Tensor*, *b: Tensor*, *add_diag_eps: bool = False*, *diag_inf: bool = False*)

L2 pairwise distance of a and b. The diagonal is either replaced with a small eps or infinite.

Examples

```
>>> import torch
>>> a = torch.tensor([[1., 2], [3, 4]])
>>> b = torch.tensor([[1., 2], [3, 4]])
>>> safe_cdist(a, b)
tensor([[0.0000, 2.8284],
        [2.8284, 0.0000]])
>>> safe_cdist(a, b, add_diag_eps=True)
```

(continues on next page)

(continued from previous page)

```
tensor([[1.4142e-12, 2.8284e+00],
        [2.8284e+00, 1.4142e-12]])
>>> safe_cdist(a, b, diag_inf=True)
tensor([[   inf, 2.8284],
        [2.8284,    inf]])
```

Parameters

- **a** (*torch.Tensor*) – First Tensor. Shape: (**BA*, na, ndim)
- **n** (*torch.Tensor*) – Second Tensor. Shape: (**BB*, nb, ndim)

Returns

Pairwise distance. Shape: (**BAB*, na, nb)

Return type

torch.Tensor

3.32 Licensing and Commercial Uses

DeepChem is licensed under the MIT License. We actively support commercial users. Note that any novel molecules, materials, or other discoveries powered by DeepChem belong entirely to the user and not to DeepChem developers.

That said, we would very much appreciate a citation if you find our tools useful. You can cite DeepChem with the following reference.

```
@book{Ramsundar-et-al-2019,
    title={Deep Learning for the Life Sciences},
    author={Bharath Ramsundar and Peter Eastman and Patrick Walters and Vijay Pande and␣
↪Karl Leswing and Zhenqin Wu},
    publisher={O'Reilly Media},
    note={\url{https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/
↪1492039837}},
    year={2019}
}
```

3.33 Contibuting to DeepChem as a Scientist

The scientific community in many ways is quite traditional. Students typically learn in apprenticeship from advisors who teach a small number of students directly. This system has endured for centuries and allows for expert scientists to teach their ways of thinking to new students.

For more context, most scientific research today is done in “labs” run in this mostly traditional fashion. A principal investigator (PI) will run the lab and work with undergraduate, graduate, and postdoctoral students who produce research papers. Labs are funded by “grants,” typically from governments and philanthropic agencies. Papers and citations are the critical currencies of this system, and a strong publication record is necessary for any scientist to establish themselves.

This traditional model can find it difficult to fund the development of high quality software for a few reasons. First, students are in a lab for limited periods of time (3-5 years often). This means there’s high turnover, and critical knowledge can be lost when a student moves on. Second, grants for software are still new and not broadly available. A lab might very reasonably choose to focus on scientific discovery rather than on necessary software engineering. (Although, it’s worth noting there are many exceptions that prove the rule! DeepChem was born in an academic lab like many other quality projects.)

We believe that contributing to and using DeepChem can be highly valuable for scientific careers. DeepChem can help maintain new scientific algorithms for the long term, making sure that your discoveries continue to be used after students graduate. We've seen too many brilliant projects flounder after students move on, and we'd like to help you make sure that your algorithms have the most impact.

3.33.1 Scientist FAQ

Contents

- *Wouldn't it be better for my career to make my own package rather than use DeepChem?*
- *Is there a DeepChem PI?*
- *Do I need to add DeepChem team members as co-authors to my paper?*
- *I want to establish my scientific niche. How can I do that as a DeepChem contributor? Won't my contribution be lost in the noise?*
- *I'm an aspiring scientist, not part of a lab. Can I join DeepChem?*
- *Is there DeepChem Grant Money?*
- *I'm an industry researcher. Can I participate too?*
- *What about intellectual property?*
- *If I use DeepChem on my organization's data, do I have to release the data?*
- *What if I want to release data? Can DeepChem help?*
- *Is MoleculeNet just about molecules?*
- *Does MoleculeNet allow for releasing data under different licenses?*

Wouldn't it be better for my career to make my own package rather than use DeepChem?

The answer to this really depends on what you're looking for out of your career! Making and maintaining good software is hard. It requires careful testing and continued maintenance. Your code will bitrot over time without attention. If your focus is on new inventions and you find software engineering less compelling, working with DeepChem may enable you to go further in your career by letting you focus on new algorithms and leveraging the DeepChem Project's infrastructure to maintain your inventions.

In addition, you may find considerable inspiration from participating in the DeepChem community. Looking at how other scientists solve problems, and connecting with new collaborators across the world can help you look at problems in a new way. Longtime DeepChem contributors find that they often end up writing papers together!

All that said, there may be very solid reasons for you to build your own project! Especially if you want to explore designs that we haven't or can't easily. In that case, we'd still love to collaborate with you. DeepChem depends on a broad constellation of scientific packages and we'd love to make your package's features accessible to our users.

Is there a DeepChem PI?

While DeepChem was born in the Pande lab at Stanford, the project now lives as a "decentralized research organization." It would be more accurate to say that there are informally multiple "DeepChem PIs," who use it in their work. You too can be a DeepChem PI!

Do I need to add DeepChem team members as co-authors to my paper?

Our suggestion is to use good judgment and usual scientific etiquette. If a particular DeepChem team member has contributed a lot to your effort, adding them might make sense. If no one person has contributed sufficiently, an acknowledgment or citation would be great!

I want to establish my scientific niche. How can I do that as a DeepChem contributor? Won't my contribution be lost in the noise?

It's critically important for a new scientist to establish themselves and their contributions in order to launch a scientific career. We believe that DeepChem can help you do this! If you add a significant set of new features to DeepChem, it might be appropriate for you to write a paper (as lead or corresponding author or however makes sense) that introduces the new feature and your contribution.

As a decentralized research organization, we want to help you launch your careers. We're very open to other collaboration structures that work for your career needs.

I'm an aspiring scientist, not part of a lab. Can I join DeepChem?

Yes! DeepChem's core mission is to democratize the use of deep learning for the sciences. This means no barriers, no walls. Anyone is welcome to join and contribute. Join our developer calls, chat one-on-one with our scientists, many of whom are glad to work with new students. You may form connections that help you join a more traditional lab, or you may choose to form your own path. We're glad to support either.

Is there DeepChem Grant Money?

Not yet, but we're actively looking into getting grants to support DeepChem researchers. If you're a PI who wants to collaborate with us, please get in touch!

I'm an industry researcher. Can I participate too?

Yes! The most powerful features of DeepChem is its community. Becoming part of the DeepChem project can let you build a network that lasts across jobs and roles. Lifelong employment at a corporation is less and less common. Joining our community will let you build bonds that cross jobs and could help you do your job today better too!

What about intellectual property?

One of the core goals for DeepChem is to build a shared set of scientific resources and techniques that aren't locked up by patents. Our hope is to enable your company or organization to leverage techniques with less worry about patent infringement.

We ask in return that you act as a responsible community member and put in as much as you get out. If you find DeepChem very valuable, please consider contributing back some innovations or improvements so others can benefit. If you're getting a patent on your invention, try to make sure that you don't infringe on anything in DeepChem. Lots of things sneak past patent review. As an open source community, we don't have the resources to actively defend ourselves and we rely on your good judgment and help!

If I use DeepChem on my organization's data, do I have to release the data?

Not at all! DeepChem is released with a permissive MIT license. Any analyses you perform belong entirely to you. You are under no obligation to release your proprietary data or inventions.

What if I want to release data? Can DeepChem help?

If you are interested in open sourcing data, the DeepChem project maintains the [MoleculeNet](https://deepchem.readthedocs.io/en/latest/moleculenet.html) suite of datasets. Adding your dataset to MoleculeNet can be a powerful way to ensure that a broad community of users can access your released data in convenient fashion. It's important to note that MoleculeNet provides programmatic access to data, which may not be appropriate for all types of data (especially for clinical or patient data which may be governed by regulations/laws). Open source datasets can be a powerful resource, but need to be handled with care.

Is MoleculeNet just about molecules?

Not anymore! Any scientific datasets are welcome in MoleculeNet. At some point in the future, we may rename the effort to avoid confusion, but for now, we emphasize that non-molecular datasets are welcome too.

Does MoleculeNet allow for releasing data under different licenses?

MoleculeNet already supports datasets released under different licenses. We can make work with you to use your license of choice.

3.34 Coding Conventions

3.34.1 Pre-Commit

We use pre-commit to ensure that we're always keeping up with the best practices when it comes to linting, standard code conventions and type annotations. Although it may seem time consuming at first as to why is one supposed to run all these tests and checks but it helps in identifying simple issues before submission to code review. We've already specified a configuration file with a list of hooks that will get executed before every commit.

First you'll need to setup the git hook scripts by installing them.

```
pre-commit install
```

Now whenever you commit, pre-commit will run the necessary hooks on the modified files.

3.34.2 Code Formatting

We use YAPF to format all of the code in DeepChem. Although it sometimes produces slightly awkward formatting, it does have two major benefits. First, it ensures complete consistency throughout the entire codebase. And second, it avoids disagreements about how a piece of code should be formatted.

Whenever you modify a file, run `yapf` on it to reformat it before checking it in.

```
yapf -i <modified file>
```

YAPF is run on every pull request to make sure the formatting is correct, so if you forget to do this the continuous integration system will remind you. Because different versions of YAPF can produce different results, it is essential to use the same version that is being run on CI. At present, that is 0.32. We periodically update it to newer versions.

3.34.3 Linting

We use Flake8 to check our code syntax. Lint tools basically provide these benefits.

- Prevent things like syntax errors or typos
- Save our review time (no need to check unused codes or typos)

Whenever you modify a file, run `flake8` on it.

```
flake8 <modified file> --count
```

If the command returns 0, it means your code passes the Flake8 check.

3.34.4 Docstrings

All classes and functions should include docstrings describing their purpose and intended usage. When in doubt about how much information to include, always err on the side of including more rather than less. Explain what problem a class is intended to solve, what algorithms it uses, and how to use it correctly. When appropriate, cite the relevant publications.

All docstrings should follow the numpy docstring formatting conventions. To ensure that the code examples in the docstrings are working as expected, run

```
python -m doctest <modified file>
```

3.34.5 Unit Tests

Having an extensive collection of test cases is essential to ensure the code works correctly. If you haven't written tests for a feature, that means the feature isn't finished yet. Untested code is code that probably doesn't work.

Complex numerical code is sometimes challenging to fully test. When an algorithm produces a result, it sometimes is not obvious how to tell whether the result is correct or not. As far as possible, try to find simple examples for which the correct answer is exactly known. Sometimes we rely on stochastic tests which will *probably* pass if the code is correct and *probably* fail if the code is broken. This means these tests are expected to fail a small fraction of the time. Such tests can be marked with the `@flaky` annotation. If they fail during continuous integration, they will be run a second time and an error only reported if they fail again.

If possible, each test should run in no more than a few seconds. Occasionally this is not possible. In that case, mark the test with the `@pytest.mark.slow` annotation. Slow tests are skipped during continuous integration, so changes that break them may sometimes slip through and get merged into the repository. We still try to run them regularly, so hopefully the problem will be discovered fairly soon.

The full suite of slow tests can be run from the root directory of the source code as

```
pytest -v -m 'slow' deepchem
```

To test your code locally, you will have to setup a symbolic link to your current development directory. To do this, simply run

```
python setup.py develop
```

while installing the package from source. This will let you see changes that you make to the source code when you import the package and, in particular, it allows you to import the new classes/methods for unit tests.

Ensure that the tests pass locally! Check this by running

```
python -m pytest <modified file>
```

3.34.6 Testing Machine Learning Models

Testing the correctness of a machine learning model can be quite tricky to do in practice. When adding a new machine learning model to DeepChem, you should add at least a few basic types of unit tests:

- Overfitting test: Create a small synthetic dataset and test that your model can learn this datasest with high accuracy. For regression and classification task, this should correspond to low training error on the dataset. For generative tasks, this should correspond to low training loss on the dataset.
- Reloading test: Check that a trained model can be saved to disk and reloaded correctly. This should involve checking that predictions from the saved and reloaded models matching exactly.

Note that unit tests are not sufficient to gauge the real performance of a model. You should benchmark your model on larger datasets as well and report your benchmarking tests in the PR comments.

For testing tensorflow models and pytorch models, we recommend testing in different conda environments. Tensorflow 2.6 supports numpy 1.19 while pytorch supports numpy 1.21. This version mismatch on numpy dependency sometimes causes trouble in installing tensorflow and pytorch backends in the same environment.

For testing tensorflow models of deepchem, we create a tensorflow test environment and then run the test as follows:

```
conda create -n tf-test python=3.8
conda activate tf-test
pip install conda-merge
conda-merge requirements/tensorflow/env_tensorflow.yml requirements/env_test.yml > env.
↪yml
conda env update --file env.yml --prune
pytest -v -m 'tensorflow' deepchem
```

For testing pytorch models of deepchem, first create a pytorch test environment and then run the tests as follows:

```
conda create -n pytorch-test python=3.8
conda activate pytorch-test
pip install conda-merge
conda-merge requirements/torch/env_torch.yml requirements/torch/env_torch.cpu.yml␣
↪requirements/env_test.yml > env.yml
conda env update --file env.yml --prune
pytest -v -m 'torch' deepchem
```

3.34.7 Type Annotations

Type annotations are an important tool for avoiding bugs. All new code should provide type annotations for function arguments and return types. When you make significant changes to existing code that does not have type annotations, please consider adding them at the same time.

We use the mypy static type checker to verify code correctness. It is automatically run on every pull request. If you want to run it locally to make sure you are using types correctly before checking in your code, `cd` to the top level directory of the repository and execute the command

```
mypy -p deepchem --ignore-missing-imports
```

Because Python is such a dynamic language, it sometimes is not obvious what type to specify. A good rule of thumb is to be permissive about input types and strict about output types. For example, many functions are documented as taking a list as an argument, but actually work just as well with a tuple. In those cases, it is best to specify the input type as `Sequence` to accept either one. But if a function returns a list, specify the type as `List` because we can guarantee the return value will always have that exact type.

Another important case is NumPy arrays. Many functions are documented as taking an array, but actually can accept any array-like object: a list of numbers, a list of lists of numbers, a list of arrays, etc. In that case, specify the type as `Sequence` to accept any of these. On the other hand, if the function truly requires an array and will fail with any other input, specify it as `np.ndarray`.

The `deepchem.utils.typing` module contains definitions of some types that appear frequently in the DeepChem API. You may find them useful when annotating code.

3.35 Understanding DeepChem CI

Continuous Integration(CI) is used to continuously build and run tests for the code in your repository to make sure that the changes introduced by the commits doesn't introduce errors. DeepChem runs a number of CI tests(jobs) using workflows provided by Github Actions. When all CI tests in a workflow pass, it implies that the changes introduced by a commit does not introduce any errors.

When creating a PR to master branch or when pushing to master branch, around 35 CI tests are run from the following workflows.

1. **Tests for DeepChem Core - The jobs are defined in the `.github/workflows/main.yml` file. The following jobs are performed in this workflow:**
    - Building and installation of DeepChem in latest Ubuntu OS and Python 3.8-3.11 and it checks for `import deepchem`
    - These tests run on Ubuntu latest version using Python 3.8-3.11 and on windows latest version using Python 3.8. The jobs are run for checking coding conventions using yapf, flake8 and mypy. It also includes tests for doctest and code-coverage.
    - Tests for pypi-build and docker-build are also include but they are mostly skipped.
2. **Tests for DeepChem Common - The jobs are defined in the `.github/workflows/common_setup.yml` file. The following tests are performed in this workflow:**
    - For build environments of Python 3.8, 3.9, 3.10, 3.11, DeepChem is built and import checking is performed.
    - The tests are run for checking pytest. All pytests which are not marked as jax, tensorflow or pytorch is run on ubuntu latest with Python 3.8, 3.9, 3.10, 3.11 and 3.9 and on windows latest, it is run with Python 3.9.
3. **Tests for DeepChem Jax/Tensorflow/PyTorch**

   - Jax - DeepChem with jax backend is installed and import check is performed for deepchem and jax. The tests for pytests with jax markers are run on ubuntu latest with Python 3.9-3.11.
   - Tensorflow - DeepChem with tensorflow backend is installed and import check is performed for DeepChem and tensorflow. The tests for pytests with tensorflow markers are run on ubuntu latest with Python 3.8-3.11 and on windows latest, it is run with Python 3.9.
   - PyTorch - DeepChem with pytorch backend is installed and import check is performed for DeepChem and torch. The tests for pytests with pytorch markers are run on ubuntu latest with Python 3.8-3.11 and on windows latest, it is run with Python 3.9.

4. **Tests for documents**
   - These tests are used for checking docs build. It is run on ubuntu latest with Python 3.9.
5. **Tests for Release**
   - These tests are run only when pushing a tag. It is run on ubuntu latest with Python 3.9.

General recommendations

1. Handling additional or external files in unittest

When a new feature is added to DeepChem, the respective unittest should included too. Sometimes, this test functions uses an external or additional file. To avoid problems in the CI the absolute path of the file has to be included. For example, for the use of a file called "Test_data_feature.csv", the unittest function should manage the absolute path as :

```
import os
current_dir = os.path.dirname(os.path.abspath(__file__))
data_dir = os.path.join(current_dir, "Test_data_feature.csv")
result = newFeature(data_dir)
```

3.35.1 Notes on Requirement Files

DeepChem's CI as well as installation procedures use requirement files defined in `requirements` directory. Currently, there are a number of requirement files. Their purposes are listed here. + *env_common.yml* - this file lists the scientific dependencies used by DeepChem like rdkit. + *env_ubuntu.yml* and *env_mac.yml* contain scientific dependencies which are have OS specific support. Currently, vina + *env_test.yml* - it is mostly used for the purpose of testing in development purpose. It contains the test dependencies. + The installation files in *tensorflow*, *torch* and *jax* directories contain the installation command for backend deep learning frameworks. For torch and jax, installation command is different for CPU and GPU. Hence, we use different installation files for CPU and GPU respectively.

3.36 Infrastructures

The DeepChem project maintains supporting infrastructure on a number of different services. This infrastructure is maintained by the DeepChem development team.

3.36.1 GitHub

The core DeepChem repositories are maintained in the deepchem GitHub organization. And, we use GitHub Actions to build a continuous integration pipeline.

DeepChem developers have write access to the repositories on this repo and technical steering committee members have admin access.

3.36.2 Conda Forge

The DeepChem feedstock repo maintains the build recipe for conda-forge.

3.36.3 Docker Hub

DeepChem hosts major releases and nightly docker build instances on Docker Hub.

3.36.4 PyPI

DeepChem hosts major releases and nightly builds on PyPI.

3.36.5 Amazon Web Services

DeepChem's website infrastructure is all managed on AWS through different AWS services. All DeepChem developers have access to these services through the deepchem-developers IAM role. (An IAM role controls access permissions.) At present, @rbharath is the only developer with admin access to the IAM role, but longer term we should migrate this so other folks have access to the roles.

S3

Amazon's S3 allows for storage of data on "buckets" (Think of buckets like folders.) There are two core deepchem S3 buckets:

- deepchemdata: This bucket hosts the MoleculeNet datasets, pre-featurized datasets, and pretrained models.
- deepchemforum: This bucket hosts backups for the forums. The bucket is private for security reasons. The forums themselves are hosted on a digital ocean instance that only @rbharath currently has access to. Longer term, we should migrate the forums onto AWS so all DeepChem developers can access the forums. The forums themselves are a discord instance. The forums upload their backups to this S3 bucket once a day. If the forums crash, they can be restored from the backups in this bucket.

Route 53

DNS for the deepchem.io website is handled by Route 53. The "hosted zone" deepchem.io holds all DNS information for the website.

Certificate Manager

The AWS certificate manager issues the SSL/TLS certificate for the *.deepchem.io and deepchem.io domains.

GitHub Pages

We make use of GitHub Pages to serve our static website. GitHub Pages connects to the certificate in Certificate Manager. We set CNAME for www.deepchem.io, and an A-record for deepchem.io.

The GitHub Pages repository is [deepchem/deepchem.github.io](https://github.com/deepchem/deepchem.github.io).

3.36.6 GoDaddy

The deepchem.io domain is registered with GoDaddy. If you change the name servers in AWS Route 53, you will need to update the GoDaddy record. At present, only @rbharath has access to the GoDaddy account that owns the deepchem.io domain name. We should explore how to provide access to the domain name for other DeepChem developers.

3.36.7 Digital Ocean

The forums are hosted on a digital ocean instance. At present, only @rbharath has access to this instance. We should migrate this instance onto AWS so other DeepChem developers can help maintain the forums.

BIBLIOGRAPHY

[gan1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.

[gan2] Arora et al., "Generalization and Equilibrium in Generative Adversarial Nets (GANs)" (https://arxiv.org/abs/1703.00573)

[wgan1] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." International conference on machine learning. PMLR, 2017. (https://arxiv.org/abs/1701.07875)

[wgan2] Gulrajani, Ishaan, et al. "Improved training of wasserstein gans." Advances in neural information processing systems 30 (2017). (https://arxiv.org/abs/1704.00028)

[molgan1] Nicola De Cao et al. "MolGAN: An implicit generative model for small molecular graphs", https://arxiv.org/abs/1805.11973

[realnvp1] Stimper, V., Schölkopf, B., & Hernández-Lobato, J. M. (2021). Resampling Base

[wgan2] Gulrajani, Ishaan, et al. "Improved training of wasserstein gans." Advances in neural information processing systems 30 (2017). (https://arxiv.org/abs/1704.00028)

[flow1] Kobyzev, I., Prince, S. J., & Brubaker, M. A. (2020). Normalizing flows: An introduction and review of current methods. IEEE transactions on pattern analysis and machine intelligence, 43(11), 3964-3979.

[maskedaffine1] Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2016). Density estimation using real nvp. arXiv preprint arXiv:1605.08803.

[glow] Kingma, D. P., & Dhariwal, P. (2018). Glow: Generative flow with invertible 1x1 convolutions. Advances in neural information processing systems, 31.

[weight_norm] Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. Advances in neural information processing systems, 29.

INDEX

Symbols

_Config (*class in deepchem.utils.dft_utils.config*), 718
_Jac (*class in deepchem.utils.differentiation_utils.grad*), 769
_RootFinder (*class in deepchem.utils.differentiation_utils.optimize.rootfinder*), 793
__add__() (*BaseXC method*), 691
__add__() (*LinearOperator method*), 748
__add__() (*ValGrad method*), 688
__contains__() (*CoordinateBox method*), 656
__getitem__() (*RadialGrid method*), 727
__init__() (*A2C method*), 622, 630
__init__() (*A2CLossContinuous method*), 628
__init__() (*A2CLossDiscrete method*), 624, 627
__init__() (*ANIFeat method*), 471
__init__() (*ActNorm method*), 561
__init__() (*AdaGrad method*), 300
__init__() (*Adam method*), 300
__init__() (*AdamW method*), 300
__init__() (*AddBaseXC method*), 696
__init__() (*AddLinearOperator method*), 752
__init__() (*AdjointLinearOperator method*), 754
__init__() (*Affine method*), 503, 558
__init__() (*AtomCGTOBasis method*), 689
__init__() (*AtomConvModel method*), 368
__init__() (*AtomEncoder method*), 515
__init__() (*AtomShim method*), 654
__init__() (*AtomicConv method*), 491
__init__() (*AtomicConvFeaturizer method*), 152
__init__() (*AtomicConvModel method*), 336
__init__() (*AtomicConvolution method*), 465
__init__() (*AtomicCoordinates method*), 143
__init__() (*AttentiveFPModel method*), 364
__init__() (*AttnLSTMEmbedding method*), 454
__init__() (*BAMFeaturizer method*), 161
__init__() (*BAMLoader method*), 60
__init__() (*BPSymmetryFunctionInput method*), 144
__init__() (*BalancingTransformer method*), 261
__init__() (*BasicMolGANModel method*), 326, 412
__init__() (*BasicSmilesTokenizer method*), 166
__init__() (*BertFeaturizer method*), 174
__init__() (*BondEncoder method*), 515
__init__() (*BroydenFirst method*), 786
__init__() (*ButinaSplitter method*), 238
__init__() (*CDFTransformer method*), 256
__init__() (*CGCNNFeaturizer method*), 156
__init__() (*CGCNNModel method*), 360
__init__() (*CGTOBasis method*), 688
__init__() (*CNN method*), 353
__init__() (*CNNModule method*), 493
__init__() (*CRAMFeaturizer method*), 162
__init__() (*CRAMLoader method*), 61
__init__() (*CSVLoader method*), 47
__init__() (*ChemCeption method*), 339
__init__() (*Chemberta method*), 440
__init__() (*CircularFingerprint method*), 133
__init__() (*ClampExp method*), 556
__init__() (*ClippingTransformer method*), 251
__init__() (*CombineMeanStd method*), 459
__init__() (*ConformerGenerator method*), 650
__init__() (*ConstScaleLayer method*), 556
__init__() (*ConvMol method*), 62
__init__() (*ConvMolFeaturizer method*), 113
__init__() (*ConvexHullPocketFinder method*), 637
__init__() (*CoordinateBox method*), 656
__init__() (*CoulombFitTransformer method*), 269
__init__() (*CoulombMatrix method*), 139
__init__() (*CoulombMatrixEig method*), 141
__init__() (*DAGGather method*), 485
__init__() (*DAGLayer method*), 484
__init__() (*DAGModel method*), 322
__init__() (*DAGTransformer method*), 274
__init__() (*DCLightningModule method*), 425
__init__() (*DE2Transformation method*), 729
__init__() (*DFTYamlLoader method*), 58
__init__() (*DMPNNEncoderLayer method*), 509
__init__() (*DMPNNFeaturizer method*), 121
__init__() (*DMPNNModel method*), 388
__init__() (*DTNN method*), 543
__init__() (*DTNNEmbedding method*), 482, 525
__init__() (*DTNNGather method*), 483, 527
__init__() (*DTNNModel method*), 321, 395
__init__() (*DTNNStep method*), 482, 526

__init__() (*DataLoader method*), 72
__init__() (*Dataset method*), 67
__init__() (*DecoderRNN method*), 546
__init__() (*DiskDataset method*), 31
__init__() (*Docker method*), 641
__init__() (*DuplicateBalancingTransformer method*), 263
__init__() (*EdgeNetwork method*), 534
__init__() (*EditableModulePureFunction method*), 757
__init__() (*ElectronSampler method*), 683
__init__() (*ElementPropertyFingerprint method*), 153
__init__() (*EncoderRNN method*), 545
__init__() (*Environment method*), 619
__init__() (*Evaluator method*), 659
__init__() (*ExponentialDecay method*), 301
__init__() (*FASTALoader method*), 54
__init__() (*FASTQLoader method*), 55
__init__() (*FakeGraphGenerator method*), 681
__init__() (*FeaturizationTransformer method*), 267
__init__() (*FerminetElectronFeature method*), 549
__init__() (*FerminetEnvelope method*), 550
__init__() (*Flow method*), 557
__init__() (*FullRankMatrix method*), 790
__init__() (*GAN method*), 331, 401
__init__() (*GANModel method*), 406
__init__() (*GATModel method*), 361
__init__() (*GBDTModel method*), 287
__init__() (*GCNModel method*), 363
__init__() (*GINEncoder method*), 511
__init__() (*GNN method*), 513
__init__() (*GNNHead method*), 513
__init__() (*GNNModular method*), 376
__init__() (*GaussianProcessHyperparamOpt method*), 612
__init__() (*GeneratorEvaluator method*), 661
__init__() (*GninaPoseGenerator method*), 640
__init__() (*GradientDescent method*), 301
__init__() (*GradientPenaltyLayer method*), 529
__init__() (*GraphCNN method*), 473
__init__() (*GraphConv method*), 444
__init__() (*GraphConvModel method*), 323
__init__() (*GraphData method*), 65
__init__() (*GraphGather method*), 448
__init__() (*GraphNetwork method*), 503
__init__() (*GraphPool method*), 446
__init__() (*GroverAtomVocabPredictor method*), 567
__init__() (*GroverAtomVocabTokenizer method*), 167
__init__() (*GroverAtomVocabularyBuilder method*), 169, 171
__init__() (*GroverAttentionHead method*), 563
__init__() (*GroverBondVocabPredictor method*), 568
__init__() (*GroverBondVocabTokenizer method*), 168
__init__() (*GroverEmbedding method*), 565, 566
__init__() (*GroverFeaturizer method*), 122
__init__() (*GroverFinetune method*), 572
__init__() (*GroverFunctionalGroupPredictor method*), 569
__init__() (*GroverMPNEncoder method*), 563
__init__() (*GroverMTBlock method*), 564
__init__() (*GroverModel method*), 391
__init__() (*GroverPretrain method*), 571
__init__() (*GroverReadout method*), 574
__init__() (*GroverTransEncoder method*), 565
__init__() (*GymEnvironment method*), 620
__init__() (*HF method*), 732
__init__() (*HFEngine method*), 734
__init__() (*Highway method*), 475
__init__() (*HighwayLayer method*), 555
__init__() (*HuggingFaceFeaturizer method*), 166
__init__() (*HuggingFaceModel method*), 436
__init__() (*HyperparamOpt method*), 608
__init__() (*IRVTransformer method*), 271
__init__() (*ImageDataset method*), 42
__init__() (*ImageLoader method*), 50
__init__() (*ImageTransformer method*), 265
__init__() (*InMemoryLoader method*), 57
__init__() (*InfoGraphEncoder method*), 510
__init__() (*InfoGraphModel method*), 372
__init__() (*InfoGraphStarModel method*), 374
__init__() (*InfoMax3DModular method*), 380
__init__() (*InteratomicL2Distances method*), 444
__init__() (*IterRefLSTMEmbedding method*), 455
__init__() (*JaxModel method*), 428
__init__() (*JsonLoader method*), 52
__init__() (*KerasModel method*), 304
__init__() (*LCNNFeaturizer method*), 159
__init__() (*LCNNModel method*), 382
__init__() (*LSTMStep method*), 453
__init__() (*Lattice method*), 685
__init__() (*LibXCGGA method*), 708
__init__() (*LibXCLDA method*), 706
__init__() (*LibXCMGGA method*), 708
__init__() (*Linear method*), 575
__init__() (*LinearCosineDecay method*), 302
__init__() (*LinearMixing method*), 788
__init__() (*LinearOperator method*), 746
__init__() (*LocalGlobalDiscriminator method*), 514
__init__() (*LogM3Transformation method*), 730
__init__() (*LogTransformer method*), 253
__init__() (*LowRankMatrix method*), 789
__init__() (*MACCSKeysFingerprint method*), 131
__init__() (*MAML method*), 614, 617
__init__() (*MATEmbedding method*), 499
__init__() (*MATEncoderLayer method*), 496
__init__() (*MATFeaturizer method*), 131
__init__() (*MATGenerator method*), 500
__init__() (*MATModel method*), 384

`__init__()` (*MEGNetModel method*), 383
`__init__()` (*MLP_flow method*), 562
`__init__()` (*MPNNModel method*), 324, 370
`__init__()` (*MXMNetBesselBasisLayer method*), 541
`__init__()` (*MXMNetFeaturizer method*), 124
`__init__()` (*MXMNetGlobalMessagePassing method*), 540
`__init__()` (*MXMNetLocalMessagePassing method*), 553
`__init__()` (*MXMNetSphericalBasisLayer method*), 554
`__init__()` (*MaskedAffineFlow method*), 560
`__init__()` (*MatmulLinearOperator method*), 755
`__init__()` (*MatrixLinearOperator method*), 756
`__init__()` (*MessagePassing method*), 486
`__init__()` (*Metric method*), 605
`__init__()` (*MinMaxTransformer method*), 249
`__init__()` (*Model method*), 283
`__init__()` (*ModularTorchModel method*), 350
`__init__()` (*Mol2VecFingerprint method*), 135
`__init__()` (*MolGANAggregationLayer method*), 450, 531
`__init__()` (*MolGANConvolutionLayer method*), 449, 530
`__init__()` (*MolGANEncoderLayer method*), 453, 533
`__init__()` (*MolGANMultiConvolutionLayer method*), 451, 532
`__init__()` (*MolGanFeaturizer method*), 116
`__init__()` (*MolGraphConvFeaturizer method*), 118
`__init__()` (*MolecularFeaturizer method*), 206
`__init__()` (*MolecularFragment method*), 653
`__init__()` (*MoleculeLoadException method*), 651
`__init__()` (*MordredDescriptors method*), 138
`__init__()` (*MulBaseXC method*), 698
`__init__()` (*MulLinearOperator method*), 753
`__init__()` (*MultiConvMol method*), 63
`__init__()` (*MultiHeadedMATAttention method*), 497
`__init__()` (*MultilayerPerceptron method*), 492
`__init__()` (*MultitaskClassifier method*), 358
`__init__()` (*MultitaskFitTransformRegressor method*), 356
`__init__()` (*MultitaskRegressor method*), 355
`__init__()` (*NTXentMultiplePositives method*), 299
`__init__()` (*NeighborList method*), 463
`__init__()` (*Net3D method*), 524
`__init__()` (*Net3DLayer method*), 521
`__init__()` (*NormalizationTransformer method*), 247
`__init__()` (*NormalizingFlowModel method*), 340
`__init__()` (*NumpyDataset method*), 25
`__init__()` (*OneHotFeaturizer method*), 147
`__init__()` (*Optimizer method*), 299
`__init__()` (*PFMFeaturizer method*), 173
`__init__()` (*PINNModel method*), 433
`__init__()` (*PNA method*), 520
`__init__()` (*PNAGNN method*), 519
`__init__()` (*PNALayer method*), 517
`__init__()` (*PPO method*), 625, 634
`__init__()` (*PPOLoss method*), 627, 637
`__init__()` (*PagtnModel method*), 366
`__init__()` (*PagtnMolGraphFeaturizer method*), 120
`__init__()` (*Policy method*), 621
`__init__()` (*PolynomialDecay method*), 301
`__init__()` (*PositionwiseFeedForward method*), 499
`__init__()` (*PowerTransformer method*), 258
`__init__()` (*ProgressiveMultitaskClassifier method*), 314
`__init__()` (*ProgressiveMultitaskModel method*), 420
`__init__()` (*ProgressiveMultitaskRegressor method*), 316
`__init__()` (*PubChemFingerprint method*), 134
`__init__()` (*PureFunction method*), 756, 759
`__init__()` (*RDKitDescriptors method*), 137
`__init__()` (*RMSProp method*), 301
`__init__()` (*RadialGrid method*), 726
`__init__()` (*RandomGroupSplitter method*), 213
`__init__()` (*RawFeaturizer method*), 149
`__init__()` (*RdkitGridFeaturizer method*), 150
`__init__()` (*RealNVPLayer method*), 505
`__init__()` (*RobertaFeaturizer method*), 175
`__init__()` (*RobustMultitaskClassifier method*), 312
`__init__()` (*RobustMultitaskRegressor method*), 313
`__init__()` (*RxnFeaturizer method*), 203
`__init__()` (*RxnSplitTransformer method*), 276
`__init__()` (*SAMFeaturizer method*), 160
`__init__()` (*SAMLoader method*), 59
`__init__()` (*SCF_QCCalc method*), 739
`__init__()` (*SDFLoader method*), 53
`__init__()` (*SNAPFeaturizer method*), 150
`__init__()` (*ScScoreModel method*), 328
`__init__()` (*ScaleNorm method*), 495
`__init__()` (*ScaledDotProductAttention method*), 572
`__init__()` (*SelfAttention method*), 573
`__init__()` (*SeqToSeq method*), 329, 548
`__init__()` (*SeqToSeqModel method*), 397
`__init__()` (*SetGather method*), 489, 512
`__init__()` (*SineCoulombMatrix method*), 155
`__init__()` (*SingletaskStratifiedSplitter method*), 219
`__init__()` (*SklearnModel method*), 286
`__init__()` (*SlicedRadialGrid method*), 728
`__init__()` (*Smiles2Vec method*), 337
`__init__()` (*SmilesToImage method*), 146
`__init__()` (*SmilesToSeq method*), 145
`__init__()` (*SmilesTokenizer method*), 164
`__init__()` (*SparseAdam method*), 300
`__init__()` (*SparseMatrixOneHotFeaturizer method*), 148
`__init__()` (*SpecifiedSplitter method*), 225
`__init__()` (*SpinParam method*), 687

__init__() (*SublayerConnection method*), 498
__init__() (*TaskSplitter method*), 227
__init__() (*TensorflowMultitaskIRVClassifier method*), 311
__init__() (*TextCNNModel method*), 335, 422
__init__() (*TorchModel method*), 342
__init__() (*TorchNNPureFunction method*), 758
__init__() (*Transformer method*), 278
__init__() (*TreutlerM4Transformation method*), 731
__init__() (*UNetModel method*), 425
__init__() (*Uniquifier method*), 811
__init__() (*UserCSVLoader method*), 48
__init__() (*UserDefinedFeaturizer method*), 204
__init__() (*ValGrad method*), 688
__init__() (*VariationalRandomizer method*), 544
__init__() (*VinaPoseGenerator method*), 639
__init__() (*WGAN method*), 334
__init__() (*WGANModel method*), 410
__init__() (*Weave method*), 415
__init__() (*WeaveFeaturizer method*), 115
__init__() (*WeaveGather method*), 480, 538
__init__() (*WeaveLayer method*), 478, 536
__init__() (*WeaveModel method*), 318, 417
__init__() (*WeaveMol method*), 64
__init__() (*WeightedLinearCombo method*), 457
__init__() (*_Config method*), 719
__init__() (*_Jac method*), 769
__len__() (*Dataset method*), 67
__len__() (*DiskDataset method*), 39
__len__() (*ImageDataset method*), 42
__len__() (*NumpyDataset method*), 25
__len__() (*RobertaFeaturizer method*), 175
__module__ (*GraphConvConstants attribute*), 125
__module__ (*MultiConvMol attribute*), 63
__module__ (*WeaveMol attribute*), 64
__mul__() (*BaseXC method*), 692
__mul__() (*ValGrad method*), 688
__new__() (*LinearOperator static method*), 745
__repr__() (*FingerprintSplitter method*), 241
__repr__() (*IndexSplitter method*), 222
__repr__() (*MaxMinSplitter method*), 236
__repr__() (*MolecularWeightSplitter method*), 233
__repr__() (*RandomSplitter method*), 210
__repr__() (*RandomStratifiedSplitter method*), 216
__repr__() (*ScaffoldSplitter method*), 230
__rmul__() (*BaseXC method*), 694
__str__() (*FingerprintSplitter method*), 242
__str__() (*IndexSplitter method*), 222
__str__() (*MaxMinSplitter method*), 236
__str__() (*MolecularWeightSplitter method*), 233
__str__() (*RandomSplitter method*), 211
__str__() (*RandomStratifiedSplitter method*), 217
__str__() (*ScaffoldSplitter method*), 230
__sub__() (*LinearOperator method*), 749
__weakref__ (*FingerprintSplitter attribute*), 242
__weakref__ (*IndexSplitter attribute*), 222
__weakref__ (*MaxMinSplitter attribute*), 236
__weakref__ (*MolecularWeightSplitter attribute*), 233
__weakref__ (*RandomSplitter attribute*), 211
__weakref__ (*RandomStratifiedSplitter attribute*), 217
__weakref__ (*ScaffoldSplitter attribute*), 230
_check_degen() (*in module deepchem.utils.differentiation_utils.symeig*), 767
_check_identical_objs() (*in module deepchem.utils.differentiation_utils.pure_function*), 759
_family (*LibXCGGA attribute*), 707
_family (*LibXCLDA attribute*), 706
_family (*LibXCMGGA attribute*), 708
_get_equilibrium_default_method() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 794
_get_minimizer_default_method() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 795
_get_rootfinder_default_method() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 794
_nonlin_solver() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 782
_nonline_line_search() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 785
_polfcn_wrapper (*LibXCGGA attribute*), 708
_polfcn_wrapper (*LibXCLDA attribute*), 706
_polfcn_wrapper (*LibXCMGGA attribute*), 708
_rootfinder_solve() (*in module deepchem.utils.differentiation_utils.solve*), 778
_safe_norm() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 785
_scalar_search_armijo() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 785
_set_initial_v() (*in module deepchem.utils.differentiation_utils.symeig*), 762
_setup_idxs() (*in module deepchem.utils.differentiation_utils.grad*), 769
_take_eigpairs() (*in module deepchem.utils.differentiation_utils.symeig*), 763
_unpolfcn_wrapper (*LibXCGGA attribute*), 707
_unpolfcn_wrapper (*LibXCLDA attribute*), 706

_unpolfcn_wrapper (*LibXCMGGA attribute*), 708

A

A2C (*class in deepchem.rl.a2c*), 621
A2C (*class in deepchem.rl.torch_rl*), 628
A2CLossContinuous (*class in deepchem.rl.torch_rl*), 627
A2CLossDiscrete (*class in deepchem.rl.a2c*), 624
A2CLossDiscrete (*class in deepchem.rl.torch_rl*), 627
accuracy_score() (*in module deepchem.metrics*), 597
action_shape (*Environment property*), 620
ActNorm (*class in deepchem.models.torch_models.flows*), 560
AdaGrad (*class in deepchem.models.optimizers*), 299
Adam (*class in deepchem.models.optimizers*), 300
adam() (*in module deepchem.utils.differentiation_utils.optimize.minimizer*), 781
AdamW (*class in deepchem.models.optimizers*), 300
add_adapter() (*ProgressiveMultitaskRegressor method*), 316
add_hydrogens_to_mol() (*in module deepchem.utils.rdkit_utils*), 652
add_loss() (*VariationalRandomizer method*), 544
add_padding_tokens() (*SmilesTokenizer method*), 165
add_shard() (*DiskDataset method*), 38
add_special_tokens() (*RobertaFeaturizer method*), 175
add_special_tokens_ids_sequence_pair() (*SmilesTokenizer method*), 165
add_special_tokens_ids_single_sequence() (*SmilesTokenizer method*), 164
add_special_tokens_single_sequence() (*SmilesTokenizer method*), 164
add_tokens() (*RobertaFeaturizer method*), 176
AddBaseXC (*class in deepchem.utils.dft_utils*), 695
added_tokens_decoder (*RobertaFeaturizer property*), 177
added_tokens_encoder (*RobertaFeaturizer property*), 177
additional_special_tokens (*RobertaFeaturizer property*), 177
additional_special_tokens_ids (*RobertaFeaturizer property*), 177
AddLinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 751
AdjointLinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 753
Affine (*class in deepchem.models.torch_models.flows*), 557
Affine (*class in deepchem.models.torch_models.layers*), 503
agglomerate_mols() (*ConvMol static method*), 63
aggregate_max() (*in module deepchem.utils.graph_utils*), 667
aggregate_mean() (*in module deepchem.utils.graph_utils*), 667
aggregate_min() (*in module deepchem.utils.graph_utils*), 667
aggregate_moment() (*in module deepchem.utils.graph_utils*), 668
aggregate_std() (*in module deepchem.utils.graph_utils*), 668
aggregate_sum() (*in module deepchem.utils.graph_utils*), 668
aggregate_var() (*in module deepchem.utils.graph_utils*), 668
all_special_ids (*RobertaFeaturizer property*), 177
all_special_tokens (*RobertaFeaturizer property*), 177
all_special_tokens_extended (*RobertaFeaturizer property*), 177
AlphaShareLayer (*class in deepchem.models.layers*), 468
anderson_acc() (*in module deepchem.utils.differentiation_utils.optimize.equilibrium*), 780
angle_between() (*in module deepchem.utils.geometry_utils*), 665
angular_symmetry() (*ANIFeat method*), 472
ANIFeat (*class in deepchem.models.layers*), 471
ao_orb2dm() (*BaseHamilton method*), 716
ao_orb_params2dm() (*BaseHamilton method*), 717
aodm() (*BaseQCCalc method*), 738
aodm() (*SCF_QCCalc method*), 740
aodm2dens() (*BaseHamilton method*), 716
aoparams2dm() (*HFEngine method*), 736
aoparams2ene() (*HFEngine method*), 735
append() (*FullRankMatrix method*), 791
append() (*LowRankMatrix method*), 789
apply_chat_template() (*RobertaFeaturizer method*), 178
apply_fcn() (*SpinParam static method*), 687
as_target_tokenizer() (*RobertaFeaturizer method*), 179
assert_runtime() (*in module deepchem.utils.differentiation_utils*), 761
assertparams() (*EditableModule method*), 742
atom_features() (*in module deepchem.feat.graph_features*), 128
atom_features() (*MATFeaturizer method*), 132
atom_to_feature_vector() (*RDKitConformerFeaturizer method*), 123
atom_to_id() (*in module deepchem.feat.graph_features*), 128
atom_to_vocab() (*GroverAtomVocabularyBuilder static method*), 170, 172

atom_vocab_random_mask() (*GroverModel static method*), 392
AtomCGTOBasis (*class in deepchem.utils.dft_utils.data.datastruct*), 689
AtomConvModel (*class in deepchem.models.torch_models*), 367
AtomEncoder (*class in deepchem.models.torch_models.pna_gnn*), 514
AtomicConv (*class in deepchem.models.torch_models.layers*), 490
AtomicConvFeaturizer (*class in deepchem.feat*), 152
AtomicConvModel (*class in deepchem.models*), 335
AtomicConvolution (*class in deepchem.models.layers*), 465
AtomicCoordinates (*class in deepchem.feat*), 142
atommasses (*BaseSystem property*), 725
atompos (*BaseSystem property*), 725
AtomShim (*class in deepchem.utils.fragment_utils*), 654
atomzs (*BaseSystem property*), 725
AttentiveFPModel (*class in deepchem.models*), 364
AttnLSTMEmbedding (*class in deepchem.models.layers*), 454
auc() (*in module deepchem.metrics*), 589

B

backend_tokenizer (*RobertaFeaturizer property*), 179
backward() (*_RootFinder static method*), 794
backward() (*CalcGGALibXCPol static method*), 703
backward() (*CalcGGALibXCUnpol static method*), 701
backward() (*CalcLDALibXCPol static method*), 700
backward() (*CalcLDALibXCUnpol static method*), 701
backward() (*CalcMGGALibXCPol static method*), 705
backward() (*CalcMGGALibXCUnpol static method*), 704
backward() (*solve_torchfcn static method*), 780
backward() (*symeig_torchfcn static method*), 767
balanced_accuracy_score() (*in module deepchem.metrics*), 598
BalancingTransformer (*class in deepchem.trans*), 260
BAMFeaturizer (*class in deepchem.feat*), 161
BAMLoader (*class in deepchem.data.data_loader*), 60
BaseDF (*class in deepchem.utils.dft_utils.df.base_df*), 711
BaseGrid (*class in deepchem.utils.dft_utils*), 710
BaseGridTransform (*class in deepchem.utils.dft_utils.grid.radial_grid*), 728
BaseHamilton (*class in deepchem.utils.dft_utils.hamilton.base_hamilton*), 713
BaseOrbParams (*class in deepchem.utils.dft_utils*), 719
BaseQCCalc (*class in deepchem.utils.dft_utils.qccalc.base_qccalc*), 737
BaseSystem (*class in deepchem.utils.dft_utils.system.base_system*), 722
BaseXC (*class in deepchem.utils.dft_utils*), 689
BasicMolGANModel (*class in deepchem.models*), 325
BasicMolGANModel (*class in deepchem.models.torch_models*), 411
BasicSmilesTokenizer (*class in deepchem.feat*), 165
batch_coulomb_matrix_features() (*in module deepchem.utils.batch_utils*), 801
batch_decode() (*RobertaFeaturizer method*), 179
batch_elements() (*in module deepchem.utils.batch_utils*), 803
batch_encode_plus() (*RobertaFeaturizer method*), 179
bedroc_score() (*in module deepchem.metrics*), 603
BertFeaturizer (*class in deepchem.feat*), 174
BetaShare (*class in deepchem.models.layers*), 470
bicgstab() (*in module deepchem.utils.differentiation_utils.solve*), 779
BinaryCrossEntropy (*class in deepchem.models.losses*), 288
BindingPocketFeaturizer (*class in deepchem.feat*), 204
BindingPocketFinder (*class in deepchem.dock.binding_pocket*), 637
bond_fdim_base (*GraphConvConstants attribute*), 125
bond_features() (*in module deepchem.feat.graph_features*), 129
bond_to_feature_vector() (*RDKitConformerFeaturizer method*), 123
bond_vocab_random_mask() (*GroverModel static method*), 393
BondEncoder (*class in deepchem.models.torch_models.pna_gnn*), 515
bos_token (*RobertaFeaturizer property*), 182
bos_token_id (*RobertaFeaturizer property*), 182
BPSymmetryFunctionInput (*class in deepchem.feat*), 143
broyden1() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 783
broyden1_solve() (*in module deepchem.utils.differentiation_utils.solve*), 777
broyden2() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 783
BroydenFirst (*class in deepchem.utils.differentiation_utils.optimize.jacobian*),

786
BroydenSecond (*class in deepchem.utils.differentiation_utils.optimize.jacobian*), 787
build() (*AlphaShareLayer method*), 468
build() (*AtomicConvolution method*), 466
build() (*AttnLSTMEmbedding method*), 455
build() (*BaseDF method*), 712
build() (*BaseHamilton method*), 714
build() (*BetaShare method*), 471
build() (*DAGGather method*), 485
build() (*DAGLayer method*), 485
build() (*DTNNEmbedding method*), 482
build() (*DTNNGather method*), 484
build() (*DTNNStep method*), 483
build() (*EdgeNetwork method*), 487
build() (*GatedRecurrentUnit method*), 488
build() (*GraphCNN method*), 474
build() (*GraphConv method*), 445
build() (*GraphEmbedPoolLayer method*), 472
build() (*GroverAtomVocabularyBuilder method*), 169, 171
build() (*Highway method*), 475
build() (*IterRefLSTMEmbedding method*), 456
build() (*LSTMStep method*), 454
build() (*MessagePassing method*), 486
build() (*SetGather method*), 490
build() (*VinaFreeEnergy method*), 462
build() (*VocabularyBuilder method*), 208
build() (*WeaveGather method*), 481
build() (*WeaveLayer method*), 479
build() (*WeightedLinearCombo method*), 458
build_char_dict() (*TextCNNModel static method*), 335, 423
build_components() (*GNNModular method*), 376
build_components() (*GroverModel method*), 391
build_components() (*InfoGraphModel method*), 372
build_components() (*InfoGraphStarModel method*), 374
build_components() (*InfoMax3DModular method*), 380
build_components() (*ModularTorchModel method*), 350
build_from_csv() (*GroverAtomVocabularyBuilder method*), 169, 171
build_gnn() (*GNNModular method*), 377
build_inception_module() (*ChemCeption method*), 339
build_inputs_with_special_tokens() (*RobertaFeaturizer method*), 182
build_layers() (*MultilayerPerceptron method*), 492
build_model() (*GNNModular method*), 377
build_model() (*GroverModel method*), 392
build_model() (*InfoGraphModel method*), 372
build_model() (*InfoGraphStarModel method*), 374
build_model() (*InfoMax3DModular method*), 381
build_model() (*ModularTorchModel method*), 350
ButinaSplitter (*class in deepchem.splits*), 238

C

cached_getparamnames() (*EditableModule method*), 741
CalcGGALibXCPol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 702
CalcGGALibXCUnpol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 701
CalcLDALibXCPol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 699
CalcLDALibXCUnpol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 700
CalcMGGALibXCPol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 704
CalcMGGALibXCUnpol (*class in deepchem.utils.dft_utils.xc.libxc_wrapper*), 703
call() (*AlphaShareLayer method*), 468
call() (*ANIFeat method*), 472
call() (*AtomicConvolution method*), 466
call() (*AttnLSTMEmbedding method*), 455
call() (*BetaShare method*), 471
call() (*CombineMeanStd method*), 459
call() (*DAGGather method*), 486
call() (*DAGLayer method*), 485
call() (*DTNNEmbedding method*), 482
call() (*DTNNGather method*), 484
call() (*DTNNStep method*), 483
call() (*EdgeNetwork method*), 487
call() (*GatedRecurrentUnit method*), 488
call() (*GraphCNN method*), 474
call() (*GraphConv method*), 445
call() (*GraphEmbedPoolLayer method*), 472
call() (*GraphGather method*), 448
call() (*GraphPool method*), 447
call() (*Highway method*), 476
call() (*InteratomicL2Distances method*), 444
call() (*IterRefLSTMEmbedding method*), 456
call() (*LSTMStep method*), 454
call() (*MessagePassing method*), 486
call() (*MolGANAggregationLayer method*), 450
call() (*MolGANConvolutionLayer method*), 449
call() (*MolGANEncoderLayer method*), 453
call() (*MolGANMultiConvolutionLayer method*), 451
call() (*NeighborList method*), 463

call() (*SetGather method*), 490
call() (*SluiceLoss method*), 470
call() (*Stack method*), 461
call() (*SwitchedDropout method*), 457
call() (*VinaFreeEnergy method*), 462
call() (*WeaveGather method*), 481
call() (*WeaveLayer method*), 479
call() (*WeightedLinearCombo method*), 458
can_save_slow_tokenizer (*RobertaFeaturizer property*), 182
CategoricalCrossEntropy (*class in deepchem.models.losses*), 288
CDFTransformer (*class in deepchem.trans*), 255
center() (*CoordinateBox method*), 657
cg() (*in module deepchem.utils.differentiation_utils.solve*), 778
CGCNNFeaturizer (*class in deepchem.feat*), 156
CGCNNModel (*class in deepchem.models*), 359
CGTOBasis (*class in deepchem.utils.dft_utils.data.datastruct*), 688
change_basis_real_to_complex() (*in module deepchem.utils.equivariance_utils*), 808
charge (*BaseSystem property*), 725
Chemberta (*class in deepchem.models.torch_models.chemberta*), 439
ChemCeption (*class in deepchem.models*), 338
CHUNK_MEMORY (*_Config attribute*), 719
chunkify() (*in module deepchem.utils.pytorch_utils*), 798
CircularFingerprint (*class in deepchem.feat*), 133
ClampExp (*class in deepchem.models.torch_models.flows*), 555
clean_up_tokenization() (*RobertaFeaturizer static method*), 182
ClippingTransformer (*class in deepchem.trans*), 251
cls_token (*RobertaFeaturizer property*), 183
cls_token_id (*RobertaFeaturizer property*), 183
CNN (*class in deepchem.models*), 352
CNNModule (*class in deepchem.models.torch_models.layers*), 492
CombineMeanStd (*class in deepchem.models.layers*), 459
complete_shuffle() (*DiskDataset method*), 37
ComplexFeaturizer (*class in deepchem.feat*), 208
compute_charges() (*in module deepchem.utils.rdkit_utils*), 652
compute_features_on_batch() (*DTNNModel method*), 321
compute_features_on_batch() (*WeaveModel method*), 319, 418
compute_metric() (*Metric method*), 606
compute_model() (*MetaLearner method*), 613
compute_model_performance() (*Evaluator method*), 660
compute_model_performance() (*GeneratorEvaluator method*), 661
compute_nbr_list() (*NeighborList method*), 464
compute_saliency() (*KerasModel method*), 309
compute_saliency() (*TorchModel method*), 347
compute_singletask_metric() (*Metric method*), 606
concordance_index() (*in module deepchem.metrics*), 603
configure_optimizers() (*DCLightningModule method*), 425
ConformerGenerator (*class in deepchem.utils.conformers*), 649
connect_graph() (*in module deepchem.utils.differentiation_utils.grad*), 770
construct_hydrogen_bonding_info() (*in module deepchem.utils.molecule_feature_utils*), 673
construct_mol() (*MATFeaturizer method*), 132
construct_node_features_matrix() (*MATFeaturizer method*), 132
ConstScaleLayer (*class in deepchem.models.torch_models.flows*), 556
contains() (*CoordinateBox method*), 657
context_pred_loss_loader() (*GNNModular method*), 378
ConvergenceWarning (*class in deepchem.utils.misc_utils*), 810
convert_atom_pair_to_voxel() (*in module deepchem.utils.voxel_utils*), 671
convert_atom_to_voxel() (*in module deepchem.utils.voxel_utils*), 670
convert_bytes_to_char() (*TextCNNModel static method*), 424
convert_ids_to_tokens() (*RobertaFeaturizer method*), 183
convert_tokens_to_ids() (*RobertaFeaturizer method*), 183
convert_tokens_to_string() (*RobertaFeaturizer method*), 183
convert_tokens_to_string() (*SmilesTokenizer method*), 164
ConvexHullPocketFinder (*class in deepchem.dock.binding_pocket*), 637
ConvMol (*class in deepchem.feat.mol_graphs*), 62
ConvMolFeaturizer (*class in deepchem.feat*), 113
coord_type (*BaseGrid property*), 711
coord_type (*RadialGrid property*), 726
CoordinateBox (*class in deepchem.utils.coordinate_box_utils*), 656
copy() (*DiskDataset method*), 33
cosine_dist() (*in module deepchem.models.layers*), 501
coulomb_matrix() (*CoulombMatrix method*), 139
coulomb_matrix() (*CoulombMatrixEig method*), 141

CoulombFitTransformer (*class in deepchem.trans*), 268
CoulombMatrix (*class in deepchem.feat*), 139
CoulombMatrixEig (*class in deepchem.feat*), 141
CRAMFeaturizer (*class in deepchem.feat*), 162
CRAMLoader (*class in deepchem.data.data_loader*), 61
create_dataset() (*BAMLoader method*), 60
create_dataset() (*CRAMLoader method*), 61
create_dataset() (*CSVLoader method*), 47
create_dataset() (*DataLoader method*), 73
create_dataset() (*DFTYamlLoader method*), 58
create_dataset() (*DiskDataset static method*), 32
create_dataset() (*FASTALoader method*), 54
create_dataset() (*FASTQLoader method*), 55
create_dataset() (*ImageLoader method*), 51
create_dataset() (*InMemoryLoader method*), 57
create_dataset() (*JsonLoader method*), 53
create_dataset() (*SAMLoader method*), 59
create_dataset() (*SDFLoader method*), 53
create_dataset() (*UserCSVLoader method*), 49
create_discriminator() (*BasicMolGANModel method*), 326, 413
create_discriminator() (*GAN method*), 332
create_discriminator() (*GANModel method*), 407
create_discriminator_loss() (*GAN method*), 332, 402
create_discriminator_loss() (*WGAN method*), 334
create_discriminator_loss() (*WGANModel method*), 410
create_generator() (*BasicMolGANModel method*), 326, 413
create_generator() (*GAN method*), 331
create_generator() (*GANModel method*), 407
create_generator_loss() (*GAN method*), 332, 402
create_generator_loss() (*WGAN method*), 334
create_generator_loss() (*WGANModel method*), 410
create_input_array() (*in module deepchem.utils.batch_utils*), 803
create_model() (*Policy method*), 621
create_nll() (*NormalizingFlowModel method*), 340
create_output_array() (*in module deepchem.utils.batch_utils*), 804
create_token_type_ids_from_sequences() (*RobertaFeaturizer method*), 183
CSVLoader (*class in deepchem.data*), 46
cutoff_filter() (*in module deepchem.dock.pose_scoring*), 643

D

DAGGather (*class in deepchem.models.layers*), 485
DAGLayer (*class in deepchem.models.layers*), 484
DAGModel (*class in deepchem.models*), 322
DAGTransformer (*class in deepchem.trans*), 273
data_dir (*DiskDataset attribute*), 31
DataLoader (*class in deepchem.data*), 72
Dataset (*class in deepchem.data*), 67
davidson() (*in module deepchem.utils.differentiation_utils.symeig*), 764
DCLightningModule (*class in deepchem.models*), 425
DE2Transformation (*class in deepchem.utils.dft_utils.grid.radial_grid*), 729
decode() (*RobertaFeaturizer method*), 184
decoder (*RobertaFeaturizer property*), 184
DecoderRNN (*class in deepchem.models.torch_models.layers*), 545
DeepGraphInfomaxLoss (*class in deepchem.models.losses*), 296
default_chat_template (*RobertaFeaturizer property*), 184
default_generator() (*AtomConvModel method*), 368
default_generator() (*AtomicConvModel method*), 337
default_generator() (*ChemCeption method*), 339
default_generator() (*CNN method*), 354
default_generator() (*DAGModel method*), 322
default_generator() (*DMPNNModel method*), 389
default_generator() (*DTNNModel method*), 321, 395
default_generator() (*GNNModular method*), 378
default_generator() (*GraphConvModel method*), 323
default_generator() (*InfoGraphStarModel method*), 375
default_generator() (*JaxModel method*), 431
default_generator() (*KerasModel method*), 309
default_generator() (*MATModel method*), 386
default_generator() (*MPNNModel method*), 324
default_generator() (*MultitaskClassifier method*), 359
default_generator() (*MultitaskFitTransformRegressor method*), 357
default_generator() (*MultitaskRegressor method*), 356
default_generator() (*PINNModel method*), 433
default_generator() (*RobustMultitaskClassifier method*), 312
default_generator() (*RobustMultitaskRegressor method*), 314
default_generator() (*ScScoreModel method*), 328
default_generator() (*Smiles2Vec method*), 338
default_generator() (*TextCNNModel method*), 335, 423
default_generator() (*TorchModel method*), 347
default_generator() (*WeaveModel method*), 320, 419
defeaturize() (*MolGanFeaturizer method*), 117
degen_symeig() (*in module

deepchem.utils.differentiation_utils.symeig), 764
del_attr (*class in deepchem.utils.attribute_utils*), 796
densityfit() (*BaseSystem method*), 723
DensityProfileLoss (*class in deepchem.models.losses*), 297
descriptors (*MordredDescriptors attribute*), 138
device (*BaseGrid property*), 711
device (*HFEngine property*), 735
device (*LinearOperator property*), 750
device (*RadialGrid property*), 726
df (*BaseHamilton property*), 714
DFTYamlLoader (*class in deepchem.data.data_loader*), 58
diagonalize() (*HFEngine method*), 737
disable_state_change() (*PureFunction method*), 757, 759
discrim_loss_fn() (*GAN method*), 403
DiskDataset (*class in deepchem.data*), 30
distance_cutoff() (*ANIFeat method*), 472
distance_matrix() (*ANIFeat method*), 472
distance_matrix() (*AtomicConvolution method*), 467
distance_tensor() (*AtomicConvolution method*), 467
dm2ao_orb_params() (*BaseHamilton method*), 718
dm2energy() (*BaseQCCalc method*), 738
dm2energy() (*HFEngine method*), 736
dm2energy() (*SCF_QCCalc method*), 740
dm2scp() (*HFEngine method*), 735
DMPNNEncoderLayer (*class in deepchem.models.torch_models.layers*), 505
DMPNNFeaturizer (*class in deepchem.feat*), 120
DMPNNModel (*class in deepchem.models.torch_models*), 387
dock() (*Docker method*), 642
Docker (*class in deepchem.dock.docking*), 641
dot() (*in module deepchem.utils.differentiation_utils.solve*), 773
download_url() (*in module deepchem.utils.data_utils*), 646
DTNN (*class in deepchem.models.torch_models.dtnn*), 542
DTNNEmbedding (*class in deepchem.models.layers*), 482
DTNNEmbedding (*class in deepchem.models.torch_models.layers*), 524
DTNNGather (*class in deepchem.models.layers*), 483
DTNNGather (*class in deepchem.models.torch_models.layers*), 526
DTNNModel (*class in deepchem.models*), 320
DTNNModel (*class in deepchem.models.torch_models*), 394
DTNNStep (*class in deepchem.models.layers*), 482
DTNNStep (*class in deepchem.models.torch_models.layers*), 525
dtype (*BaseGrid property*), 710
dtype (*HFEngine property*), 734
dtype (*LinearOperator property*), 750
dtype (*RadialGrid property*), 726
dummy_context_manager() (*in module deepchem.utils.differentiation_utils*), 761
DummyFeaturizer (*class in deepchem.feat*), 205
DuplicateBalancingTransformer (*class in deepchem.trans*), 262

E

edge_features (*GraphData attribute*), 64
edge_index (*GraphData attribute*), 64
EdgeNetwork (*class in deepchem.models.layers*), 486
EdgeNetwork (*class in deepchem.models.torch_models.layers*), 534
EdgePredictionLoss (*class in deepchem.models.losses*), 293
EditableModule (*class in deepchem.utils.differentiation_utils.editable_module*), 740
EditableModulePureFunction (*class in deepchem.utils.differentiation_utils.pure_function*), 757
efield (*BaseSystem property*), 725
electron_update() (*ElectronSampler method*), 684
ElectronSampler (*class in deepchem.utils.electron_sampler*), 682
ElementPropertyFingerprint (*class in deepchem.feat*), 153
ElemNetFeaturizer (*class in deepchem.feat*), 154
embed_molecule() (*ConformerGenerator method*), 650
encode() (*GroverAtomVocabularyBuilder method*), 170, 172
encode() (*RobertaFeaturizer method*), 184
encode_bio_sequence() (*in module deepchem.utils.genomics_utils*), 663
encode_plus() (*RobertaFeaturizer method*), 186
EncoderRNN (*class in deepchem.models.torch_models.layers*), 545
energy() (*BaseQCCalc method*), 738
energy() (*SCF_QCCalc method*), 740
Environment (*class in deepchem.rl*), 619
eos_token (*RobertaFeaturizer property*), 188
eos_token_id (*RobertaFeaturizer property*), 189
equilibrium() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 792
estimate_ewald_eta() (*Lattice method*), 686
evaluate() (*Model method*), 284
evaluate_generator() (*JaxModel method*), 431
evaluate_generator() (*KerasModel method*), 308
evaluate_generator() (*TorchModel method*), 346
Evaluator (*class in deepchem.utils.evaluate*), 659
exacteig() (*in module deepchem.utils.differentiation_utils.symeig*),

764
exactsolve() (*in module deepchem.utils.differentiation_utils.solve*), 771
expand() (*CoulombFitTransformer method*), 269
ExponentialDecay (*class in deepchem.models.optimizers*), 301
extend() (*VocabularyBuilder method*), 208
extract_grover_attributes() (*in module deepchem.utils.grover*), 677

F

f1_score() (*in module deepchem.metrics*), 592
FakeGraphGenerator (*class in deepchem.utils.fake_data_generator*), 680
family (*AddBaseXC property*), 696
family (*BaseXC property*), 690
family (*LibXCLDA property*), 706
family (*MulBaseXC property*), 698
FASTALoader (*class in deepchem.data*), 54
FASTQLoader (*class in deepchem.data*), 55
features_to_id() (*in module deepchem.feat.graph_features*), 127
FeaturizationTransformer (*class in deepchem.trans*), 266
featurize() (*AtomicConvFeaturizer method*), 152
featurize() (*AtomicCoordinates method*), 143
featurize() (*BAMFeaturizer method*), 161
featurize() (*BertFeaturizer method*), 174
featurize() (*BindingPocketFeaturizer method*), 204
featurize() (*BPSymmetryFunctionInput method*), 144
featurize() (*CGCNNFeaturizer method*), 157
featurize() (*CircularFingerprint method*), 134
featurize() (*ComplexFeaturizer method*), 208
featurize() (*ConvMolFeaturizer method*), 114
featurize() (*CoulombMatrix method*), 140
featurize() (*CoulombMatrixEig method*), 141
featurize() (*CRAMFeaturizer method*), 162
featurize() (*DataLoader method*), 72
featurize() (*DMPNNFeaturizer method*), 121
featurize() (*DummyFeaturizer method*), 205
featurize() (*ElementPropertyFingerprint method*), 153
featurize() (*Featurizer method*), 205
featurize() (*HuggingFaceFeaturizer method*), 166
featurize() (*LCNNFeaturizer method*), 159
featurize() (*MaterialCompositionFeaturizer method*), 207
featurize() (*MaterialStructureFeaturizer method*), 207
featurize() (*MATFeaturizer method*), 132
featurize() (*Mol2VecFingerprint method*), 136
featurize() (*MolecularFeaturizer method*), 206
featurize() (*MolGanFeaturizer method*), 116
featurize() (*MolGraphConvFeaturizer method*), 118
featurize() (*MordredDescriptors method*), 138
featurize() (*MXMNetFeaturizer method*), 124
featurize() (*OneHotFeaturizer method*), 147
featurize() (*PagtnMolGraphFeaturizer method*), 120
featurize() (*RawFeaturizer method*), 149
featurize() (*RDKitDescriptors method*), 137
featurize() (*RdkitGridFeaturizer method*), 151
featurize() (*RobertaFeaturizer method*), 189
featurize() (*RxnFeaturizer method*), 203
featurize() (*SAMFeaturizer method*), 160
featurize() (*SineCoulombMatrix method*), 155
featurize() (*SmilesToImage method*), 146
featurize() (*SmilesToSeq method*), 145
featurize() (*SNAPFeaturizer method*), 150
featurize() (*SparseMatrixOneHotFeaturizer method*), 148
featurize() (*UserDefinedFeaturizer method*), 204
featurize() (*WeaveFeaturizer method*), 115
Featurizer (*class in deepchem.feat*), 205
FerminetElectronFeature (*class in deepchem.models.torch_models.layers*), 548
FerminetEnvelope (*class in deepchem.models.torch_models.layers*), 550
find_all_pockets() (*ConvexHullPocketFinder method*), 638
find_distance() (*in module deepchem.feat.graph_features*), 128
find_pockets() (*BindingPocketFinder method*), 637
find_pockets() (*ConvexHullPocketFinder method*), 638
FingerprintSplitter (*class in deepchem.splits*), 240
fit() (*A2C method*), 623, 631
fit() (*GBDTModel method*), 287
fit() (*JaxModel method*), 429
fit() (*KerasModel method*), 304
fit() (*MAML method*), 615, 617
fit() (*Model method*), 284
fit() (*PPO method*), 625, 634
fit() (*ProgressiveMultitaskModel method*), 420
fit() (*ProgressiveMultitaskRegressor method*), 316
fit() (*SklearnModel method*), 286
fit() (*TorchModel method*), 343
fit_gan() (*GAN method*), 332
fit_gan() (*GANModel method*), 407
fit_generator() (*HuggingFaceModel method*), 438
fit_generator() (*KerasModel method*), 305
fit_generator() (*ModularTorchModel method*), 351
fit_generator() (*TorchModel method*), 343
fit_on_batch() (*KerasModel method*), 306
fit_on_batch() (*Model method*), 283
fit_on_batch() (*TorchModel method*), 344
fit_sequences() (*SeqToSeq method*), 329
fit_sequences() (*SeqToSeqModel method*), 398

fit_task() (*ProgressiveMultitaskModel method*), 421
fit_task() (*ProgressiveMultitaskRegressor method*), 317
fit_with_eval() (*GBDTModel method*), 287
Flow (*class in deepchem.models.torch_models.flows*), 557
forward() (*_RootFinder static method*), 793
forward() (*ActNorm method*), 561
forward() (*Affine method*), 504, 558
forward() (*AtomEncoder method*), 515
forward() (*AtomicConv method*), 491
forward() (*BondEncoder method*), 516
forward() (*CalcGGALibXCPol static method*), 702
forward() (*CalcGGALibXCUnpol static method*), 701
forward() (*CalcLDALibXCPol static method*), 699
forward() (*CalcLDALibXCUnpol static method*), 700
forward() (*CalcMGGALibXCPol static method*), 704
forward() (*CalcMGGALibXCUnpol static method*), 703
forward() (*ClampExp method*), 556
forward() (*CNNModule method*), 494
forward() (*ConstScaleLayer method*), 556
forward() (*DecoderRNN method*), 546
forward() (*DMPNNEncoderLayer method*), 509
forward() (*DTNN method*), 543
forward() (*DTNNEmbedding method*), 525
forward() (*DTNNGather method*), 527
forward() (*DTNNStep method*), 526
forward() (*EdgeNetwork method*), 535
forward() (*EncoderRNN method*), 545
forward() (*FerminetElectronFeature method*), 549
forward() (*FerminetEnvelope method*), 551
forward() (*Flow method*), 557
forward() (*GAN method*), 402
forward() (*GINEncoder method*), 511
forward() (*GNN method*), 513
forward() (*GNNHead method*), 513
forward() (*GradientPenaltyLayer method*), 529
forward() (*GraphNetwork method*), 503
forward() (*GroverAtomVocabPredictor method*), 567
forward() (*GroverAttentionHead method*), 564
forward() (*GroverBondVocabPredictor method*), 568
forward() (*GroverEmbedding method*), 565, 566
forward() (*GroverFinetune method*), 572
forward() (*GroverFunctionalGroupPredictor method*), 569
forward() (*GroverMPNEncoder method*), 563
forward() (*GroverMTBlock method*), 564
forward() (*GroverPretrain method*), 571
forward() (*GroverReadout method*), 574
forward() (*GroverTransEncoder method*), 565
forward() (*HighwayLayer method*), 555
forward() (*InfoGraphEncoder method*), 510
forward() (*LocalGlobalDiscriminator method*), 514
forward() (*MaskedAffineFlow method*), 560
forward() (*MATEmbedding method*), 500
forward() (*MATEncoderLayer method*), 496
forward() (*MATGenerator method*), 500
forward() (*MLP_flow method*), 562
forward() (*MolGANAggregationLayer method*), 531
forward() (*MolGANConvolutionLayer method*), 530
forward() (*MolGANEncoderLayer method*), 534
forward() (*MolGANMultiConvolutionLayer method*), 532
forward() (*MultiHeadedMATAttention method*), 497
forward() (*MultilayerPerceptron method*), 492
forward() (*MXMNetBesselBasisLayer method*), 542
forward() (*MXMNetGlobalMessagePassing method*), 540
forward() (*MXMNetLocalMessagePassing method*), 553
forward() (*MXMNetSphericalBasisLayer method*), 554
forward() (*Net3D method*), 524
forward() (*Net3DLayer method*), 522
forward() (*PNA method*), 521
forward() (*PNAGNN method*), 519
forward() (*PNALayer method*), 517
forward() (*PositionwiseFeedForward method*), 499
forward() (*RealNVPLayer method*), 505
forward() (*ScaledDotProductAttention method*), 572
forward() (*ScaleNorm method*), 495
forward() (*SelfAttention method*), 573
forward() (*SeqToSeq method*), 548
forward() (*SetGather method*), 512
forward() (*solve_torchfcn static method*), 779
forward() (*SublayerConnection method*), 498
forward() (*symeig_torchfcn static method*), 767
forward() (*VariationalRandomizer method*), 544
forward() (*Weave method*), 416
forward() (*WeaveGather method*), 539
forward() (*WeaveLayer method*), 537
fourier_encode_dist() (*in module deepchem.utils.graph_utils*), 667
freeze_components() (*ModularTorchModel method*), 350
from_dataframe() (*Dataset static method*), 71
from_dataframe() (*DiskDataset static method*), 40
from_dataframe() (*ImageDataset static method*), 44
from_dataframe() (*NumpyDataset static method*), 28
from_DiskDataset() (*NumpyDataset static method*), 27
from_json() (*NumpyDataset static method*), 27
from_numpy() (*DiskDataset static method*), 35
from_one_hot() (*in module deepchem.metrics*), 576
from_pretrained() (*RobertaFeaturizer class method*), 189
fullmatrix() (*LinearOperator method*), 747
FullRankMatrix (*class in deepchem.utils.differentiation_utils.optimize.jacobian*), 790

FunctionPureFunction (*class in deepchem.utils.differentiation_utils.pure_function*), 757

G

GAN (*class in deepchem.models*), 330
GAN (*class in deepchem.models.torch_models*), 399
GANModel (*class in deepchem.models.torch_models*), 403
GatedRecurrentUnit (*class in deepchem.models.layers*), 488
GATModel (*class in deepchem.models*), 360
gauss_initialize_position() (*ElectronSampler method*), 684
gaussian_distance_matrix() (*AtomicConvolution method*), 467
gaussian_first() (*VinaFreeEnergy method*), 462
gaussian_histogram() (*WeaveGather method*), 481, 539
gaussian_integral() (*in module deepchem.utils.pytorch_utils*), 799
gaussian_second() (*VinaFreeEnergy method*), 462
GaussianProcessHyperparamOpt (*class in deepchem.hyper*), 611
GBDTModel (*class in deepchem.models*), 287
GCNModel (*class in deepchem.models*), 362
gd() (*in module deepchem.utils.differentiation_utils.optimize.minimizer*), 781
gen_loss_fn() (*GAN method*), 403
generate_conformers() (*ConformerGenerator method*), 650
generate_poses() (*GninaPoseGenerator method*), 641
generate_poses() (*PoseGenerator method*), 638
generate_poses() (*VinaPoseGenerator method*), 639
generate_random_rotation_matrix() (*in module deepchem.utils.geometry_utils*), 665
generate_random_unit_vector() (*in module deepchem.utils.geometry_utils*), 665
generate_scaffolds() (*ScaffoldSplitter method*), 229
GeneratorEvaluator (*class in deepchem.utils.evaluate*), 661
get_added_vocab() (*RobertaFeaturizer method*), 190
get_adjacency_list() (*ConvMol method*), 62
get_and_pop_keys() (*in module deepchem.utils.differentiation_utils*), 760
get_atom_chirality_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 674
get_atom_features() (*ConvMol method*), 62
get_atom_features() (*MultiConvMol method*), 63
get_atom_features() (*WeaveMol method*), 64
get_atom_formal_charge() (*in module deepchem.utils.molecule_feature_utils*), 674
get_atom_hybridization_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 673
get_atom_hydrogen_bonding_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 673
get_atom_is_in_aromatic_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 673
get_atom_partial_charge() (*in module deepchem.utils.molecule_feature_utils*), 674
get_atom_total_degree_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 675
get_atom_total_num_Hs_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 674
get_atom_type_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 672
get_atoms_in_nbrs() (*NeighborList method*), 464
get_atoms_with_deg() (*ConvMol method*), 62
get_atomz() (*in module deepchem.utils.periodic_table_utils*), 805
get_attr (*class in deepchem.utils.attribute_utils*), 795
get_batch() (*MetaLearner method*), 614
get_batchdims() (*in module deepchem.utils.differentiation_utils.solve*), 772
get_bcasted_dims() (*in module deepchem.utils.differentiation_utils*), 743
get_bond_graph_distance_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 676
get_bond_is_conjugated_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 676
get_bond_is_in_same_ring_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 675
get_bond_stereo_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 676
get_bond_type_one_hot() (*in module deepchem.utils.molecule_feature_utils*), 675
get_cells() (*NeighborList method*), 465
get_cells_for_atoms() (*NeighborList method*), 465
get_checkpoints() (*A2C method*), 632
get_checkpoints() (*KerasModel method*), 310
get_checkpoints() (*MAML method*), 618
get_checkpoints() (*PPO method*), 636
get_checkpoints() (*TorchModel method*), 348
get_closest_atoms() (*NeighborList method*), 464
get_conditional_input_shapes() (*GAN method*), 331
get_conditional_input_shapes() (*GANModel method*), 407
get_config() (*AlphaShareLayer method*), 468
get_config() (*ANIFeat method*), 471
get_config() (*AtomicConvolution method*), 466
get_config() (*AttnLSTMEmbedding method*), 454
get_config() (*BetaShare method*), 471
get_config() (*CombineMeanStd method*), 459
get_config() (*DAGGather method*), 485
get_config() (*DAGLayer method*), 484
get_config() (*DTNNEmbedding method*), 482

get_config() (*DTNNGather method*), 483
get_config() (*DTNNStep method*), 482
get_config() (*EdgeNetwork method*), 486
get_config() (*GatedRecurrentUnit method*), 488
get_config() (*GraphCNN method*), 473
get_config() (*GraphConv method*), 445
get_config() (*GraphEmbedPoolLayer method*), 472
get_config() (*GraphGather method*), 448
get_config() (*GraphPool method*), 446
get_config() (*Highway method*), 475
get_config() (*InteratomicL2Distances method*), 444
get_config() (*IterRefLSTMEmbedding method*), 456
get_config() (*LSTMStep method*), 454
get_config() (*MessagePassing method*), 486
get_config() (*MolGANAggregationLayer method*), 451
get_config() (*MolGANConvolutionLayer method*), 449
get_config() (*MolGANEncoderLayer method*), 453
get_config() (*MolGANMultiConvolutionLayer method*), 452
get_config() (*NeighborList method*), 463
get_config() (*SetGather method*), 489
get_config() (*SluiceLoss method*), 469
get_config() (*Stack method*), 460
get_config() (*SwitchedDropout method*), 456
get_config() (*VinaFreeEnergy method*), 462
get_config() (*WeaveGather method*), 480
get_config() (*WeaveLayer method*), 479
get_config() (*WeightedLinearCombo method*), 458
get_conformer_energies() (*ConformerGenerator method*), 651
get_conformer_rmsd() (*ConformerGenerator static method*), 651
get_contact_atom_indices() (*in module deepchem.utils.fragment_utils*), 655
get_data_dir() (*in module deepchem.utils.data_utils*), 646
get_data_input_shapes() (*BasicMolGANModel method*), 326, 413
get_data_input_shapes() (*GAN method*), 331
get_data_input_shapes() (*GANModel method*), 406
get_data_shape() (*DiskDataset method*), 33
get_deg_adjacency_lists() (*ConvMol method*), 62
get_deg_adjacency_lists() (*MultiConvMol method*), 63
get_deg_slice() (*ConvMol method*), 63
get_drdx() (*BaseGridTransform method*), 728
get_drdx() (*DE2Transformation method*), 729
get_drdx() (*LogM3Transformation method*), 730
get_drdx() (*TreutlerM4Transformation method*), 731
get_dvolume() (*BaseGrid method*), 711
get_dvolume() (*RadialGrid method*), 727
get_e_elrep() (*BaseHamilton method*), 717
get_e_exchange() (*BaseHamilton method*), 717
get_e_hcore() (*BaseHamilton method*), 716
get_e_xc() (*BaseHamilton method*), 717
get_edensityxc() (*AddBaseXC method*), 696
get_edensityxc() (*BaseXC method*), 690
get_edensityxc() (*LibXCLDA method*), 706
get_edensityxc() (*MulBaseXC method*), 699
get_elrep() (*BaseDF method*), 712
get_elrep() (*BaseHamilton method*), 715
get_exchange() (*BaseHamilton method*), 715
get_face_boxes() (*in module deepchem.utils.coordinate_box_utils*), 658
get_feature_list() (*in module deepchem.feat.graph_features*), 127
get_floor_and_ceil() (*in module deepchem.utils.safeops_utils*), 813
get_global_step() (*JaxModel method*), 431
get_global_step() (*KerasModel method*), 310
get_global_step() (*TorchModel method*), 348
get_grid() (*BaseSystem method*), 724
get_grid_transform (*class in deepchem.utils.dft_utils.grid.radial_grid*), 731
get_gvgrids() (*Lattice method*), 686
get_hamiltonian() (*BaseSystem method*), 723
get_interatomic_distances() (*CoulombMatrix static method*), 140
get_interatomic_distances() (*CoulombMatrixEig static method*), 142
get_intervals() (*in module deepchem.feat.graph_features*), 126
get_kinnucl() (*BaseHamilton method*), 714
get_label_means() (*DiskDataset method*), 40
get_label_stds() (*DiskDataset method*), 40
get_largest_eival() (*in module deepchem.utils.differentiation_utils.solve*), 776
get_lattice_ls() (*Lattice method*), 686
get_libxc() (*in module deepchem.utils.dft_utils.api.getxc*), 709
get_loss_func() (*GroverModel method*), 392
get_max_print_size() (*in module deepchem.utils.debug_utils*), 680
get_memory() (*in module deepchem.utils.pytorch_utils*), 798
get_method() (*in module deepchem.utils.differentiation_utils*), 761
get_model_filename() (*Model static method*), 283
get_molecule_force_field() (*ConformerGenerator method*), 650
get_motif_scores() (*in module deepchem.metrics.genomic_metrics*), 604
get_neighbor_cells() (*NeighborList method*), 465
get_noise_batch() (*GAN method*), 331, 402
get_noise_input_shape() (*BasicMolGANModel

method), 326, 413
get_noise_input_shape() (*GAN method*), 331
get_noise_input_shape() (*GANModel method*), 406
get_np_dtype() (*in module deepchem.utils.pytorch_utils*), 801
get_nuclattr() (*BaseHamilton method*), 714
get_nuclei_energy() (*BaseSystem method*), 724
get_null_mol() (*ConvMol static method*), 63
get_num_atoms() (*MultiConvMol method*), 63
get_num_atoms() (*WeaveMol method*), 64
get_num_atoms_with_deg() (*ConvMol method*), 62
get_num_features() (*WeaveMol method*), 64
get_num_molecules() (*MultiConvMol method*), 63
get_num_tasks() (*Model method*), 285
get_number_shards() (*DiskDataset method*), 34
get_orbweight() (*BaseSystem method*), 724
get_overlap() (*BaseHamilton method*), 715
get_pair_edges() (*WeaveMol method*), 64
get_pair_features() (*WeaveMol method*), 64
get_params_filename() (*Model static method*), 283
get_print_threshold() (*in module deepchem.utils.debug_utils*), 680
get_pssm_scores() (*in module deepchem.metrics.genomic_metrics*), 604
get_pure_function() (*in module deepchem.utils.differentiation_utils.pure_function*), 760
get_rgrid() (*BaseGrid method*), 711
get_rgrid() (*RadialGrid method*), 727
get_shape() (*Dataset method*), 67
get_shape() (*DiskDataset method*), 39
get_shape() (*ImageDataset method*), 42
get_shape() (*NumpyDataset method*), 25
get_shard() (*DiskDataset method*), 37
get_shard_ids() (*DiskDataset method*), 38
get_shard_size() (*DiskDataset method*), 34
get_shard_w() (*DiskDataset method*), 38
get_shard_y() (*DiskDataset method*), 38
get_special_tokens_mask() (*RobertaFeaturizer method*), 191
get_statistics() (*Dataset method*), 69
get_statistics() (*DiskDataset method*), 40
get_statistics() (*ImageDataset method*), 44
get_statistics() (*NumpyDataset method*), 28
get_system() (*BaseQCCalc method*), 738
get_system() (*HFEngine method*), 734
get_system() (*SCF_QCCalc method*), 739
get_task_names() (*Dataset method*), 67
get_task_names() (*DiskDataset method*), 33
get_task_names() (*ImageDataset method*), 42
get_task_names() (*NumpyDataset method*), 25
get_task_type() (*Model method*), 285
get_unique_objs() (*Uniquifier method*), 811
get_vector() (*ElemNetFeaturizer method*), 154
get_vext() (*BaseHamilton method*), 715
get_vocab() (*RobertaFeaturizer method*), 191
get_vxc() (*AddBaseXC method*), 696
get_vxc() (*BaseHamilton method*), 716
get_vxc() (*BaseXC method*), 690
get_vxc() (*LibXCLDA method*), 706
get_vxc() (*MulBaseXC method*), 698
get_xc() (*in module deepchem.utils.dft_utils.api.getxc*), 709
get_xw_integration (*class in deepchem.utils.dft_utils.grid.radial_grid*), 727
get_xyz_from_mol() (*in module deepchem.utils.rdkit_utils*), 651
GetAtomicNum() (*AtomShim method*), 654
GetAtoms() (*MolecularFragment method*), 653
GetCoords() (*AtomShim method*), 654
GetCoords() (*MolecularFragment method*), 654
getlinopparams() (*LinearOperator method*), 746
GetNumAtoms() (*MolecularFragment method*), 654
getparamnames() (*AddBaseXC method*), 697
getparamnames() (*BaseDF method*), 712
getparamnames() (*BaseGrid method*), 711
getparamnames() (*BaseHamilton method*), 718
getparamnames() (*BaseQCCalc method*), 739
getparamnames() (*BaseSystem method*), 724
getparamnames() (*BaseXC method*), 691
getparamnames() (*EditableModule method*), 742
getparamnames() (*HFEngine method*), 737
getparamnames() (*LibXCLDA method*), 707
getparamnames() (*LinearOperator method*), 747
getparamnames() (*MulBaseXC method*), 699
getparamnames() (*RadialGrid method*), 727
getparams() (*EditableModule method*), 741
GetPartialCharge() (*AtomShim method*), 654
GetSetParamsError (*class in deepchem.utils.misc_utils*), 810
getuniqueparams() (*EditableModule method*), 742
GINEncoder (*class in deepchem.models.torch_models*), 511
GlobalMutualInformationLoss (*class in deepchem.models.losses*), 290
gmres() (*in module deepchem.utils.differentiation_utils.solve*), 773
GninaPoseGenerator (*class in deepchem.dock.pose_generation*), 640
GNN (*class in deepchem.models.torch_models.gnn*), 512
GNNHead (*class in deepchem.models.torch_models.gnn*), 513
GNNModular (*class in deepchem.models.torch_models.gnn*), 375
GradientDescent (*class in deepchem.models.optimizers*), 301

GradientPenaltyLayer (*class in deepchem.models.torch_models.gan*), 527
GraphCNN (*class in deepchem.models.layers*), 473
GraphContextPredLoss (*class in deepchem.models.losses*), 297
GraphConv (*class in deepchem.models.layers*), 444
GraphConvConstants (*class in deepchem.feat.graph_features*), 125
GraphConvModel (*class in deepchem.models*), 323
GraphData (*class in deepchem.feat.graph_data*), 64
GraphEdgeMaskingLoss (*class in deepchem.models.losses*), 295
GraphEmbedPoolLayer (*class in deepchem.models.layers*), 472
GraphGather (*class in deepchem.models.layers*), 447
GraphNetwork (*class in deepchem.models.torch_models.layers*), 502
GraphNodeMaskingLoss (*class in deepchem.models.losses*), 294
GraphPool (*class in deepchem.models.layers*), 446
GridHyperparamOpt (*class in deepchem.hyper*), 609
GroverAtomVocabPredictor (*class in deepchem.models.torch_models.grover_layers*), 566
GroverAtomVocabTokenizer (*class in deepchem.feat.vocabulary_builders.grover_vocab*), 167
GroverAtomVocabularyBuilder (*class in deepchem.feat.vocabulary_builders.grover_vocab*), 168, 171
GroverAttentionHead (*class in deepchem.models.torch_models.grover_layers*), 563
GroverBondVocabPredictor (*class in deepchem.models.torch_models.grover_layers*), 567
GroverBondVocabTokenizer (*class in deepchem.feat.vocabulary_builders.grover_vocab*), 168
GroverEmbedding (*class in deepchem.models.torch_models.grover_layers*), 565, 566
GroverFeaturizer (*class in deepchem.feat*), 122
GroverFinetune (*class in deepchem.models.torch_models.grover*), 571
GroverFunctionalGroupPredictor (*class in deepchem.models.torch_models.grover_layers*), 568
GroverModel (*class in deepchem.models.torch_models*), 390
GroverMPNEncoder (*class in deepchem.models.torch_models.grover_layers*), 563
GroverMTBlock (*class in deepchem.models.torch_models.grover_layers*), 564
GroverPretrain (*class in deepchem.models.torch_models.grover*), 569
GroverPretrainLoss (*class in deepchem.models.losses*), 292
GroverReadout (*class in deepchem.models.torch_models.readout*), 573
GroverTransEncoder (*class in deepchem.models.torch_models.grover_layers*), 564
GymEnvironment (*class in deepchem.rl*), 620

H

H (*AdjointLinearOperator property*), 754
H (*LinearOperator property*), 747
handle_classification_mode() (*in module deepchem.metrics*), 578
harmonic_mean() (*ElectronSampler method*), 683
hash_ecfp() (*in module deepchem.utils.hash_utils*), 669
hash_ecfp_pair() (*in module deepchem.utils.hash_utils*), 670
HF (*class in deepchem.utils.dft_utils.qccalc.hf*), 732
hf_vocab (*in module deepchem.feat.vocabulary_builders*), 209
HFEngine (*class in deepchem.utils.dft_utils.qccalc.hf*), 732
hhblits() (*in module deepchem.utils.sequence_utils*), 663
hhsearch() (*in module deepchem.utils.sequence_utils*), 664
Highway (*class in deepchem.models.layers*), 475
HighwayLayer (*class in deepchem.models.torch_models.layers*), 555
HingeLoss (*class in deepchem.models.losses*), 288
HuberLoss (*class in deepchem.models.losses*), 288
HuggingFaceFeaturizer (*class in deepchem.feat.huggingface_featurizer*), 166
HuggingFaceModel (*class in deepchem.models.torch_models.hf_models*), 434
hydrogen_bond() (*VinaFreeEnergy method*), 462
hydrophobic() (*VinaFreeEnergy method*), 462
hyperparam_search() (*GaussianProcessHyperparamOpt method*), 612
hyperparam_search() (*GridHyperparamOpt method*), 610
hyperparam_search() (*HyperparamOpt method*), 608
HyperparamOpt (*class in deepchem.hyper*), 608

I

id_to_features() (*in module deepchem.feat.graph_features*), 127
ids (*Dataset property*), 68
ids (*DiskDataset property*), 39
ids (*ImageDataset property*), 42
ids (*NumpyDataset property*), 25
ImageDataset (*class in deepchem.data*), 42
ImageLoader (*class in deepchem.data*), 49
ImageTransformer (*class in deepchem.trans*), 265
in_silico_mutagenesis() (*in module deepchem.metrics.genomic_metrics*), 604
indent() (*in module deepchem.utils.misc_utils*), 810
IndexSplitter (*class in deepchem.splits*), 221
InfoGraphEncoder (*class in deepchem.models.torch_models*), 510
InfoGraphModel (*class in deepchem.models.torch_models*), 370
InfoGraphStarModel (*class in deepchem.models.torch_models*), 373
InfoMax3DModular (*class in deepchem.models.torch_models.gnn3d*), 378
infomax_loss_loader() (*GNNModular method*), 377
InMemoryLoader (*class in deepchem.data*), 56
input_edge_func() (*Net3D method*), 524
InteratomicL2Distances (*class in deepchem.models.layers*), 443
intersect_interval() (*in module deepchem.utils.coordinate_box_utils*), 658
intervals (*GraphConvConstants attribute*), 125
inverse() (*ActNorm method*), 561
inverse() (*Affine method*), 504, 558
inverse() (*Flow method*), 557
inverse() (*MaskedAffineFlow method*), 560
inverse() (*RealNVPLayer method*), 505
IRVTransformer (*class in deepchem.trans*), 271
is_angle_within_cutoff() (*in module deepchem.utils.geometry_utils*), 666
is_fullmatrix_implemented (*LinearOperator property*), 751
is_getparamnames_implemented (*LinearOperator property*), 751
is_hermitian (*LinearOperator property*), 750
is_mm_implemented (*LinearOperator property*), 750
is_mv_implemented (*LinearOperator property*), 750
is_rmm_implemented (*LinearOperator property*), 751
is_rmv_implemented (*LinearOperator property*), 750
iterbatches() (*Dataset method*), 68
iterbatches() (*DiskDataset method*), 34
iterbatches() (*ImageDataset method*), 42
iterbatches() (*NumpyDataset method*), 25
IterRefLSTMEmbedding (*class in deepchem.models.layers*), 455
itersamples() (*Dataset method*), 69
itersamples() (*DiskDataset method*), 34
itersamples() (*ImageDataset method*), 43
itersamples() (*NumpyDataset method*), 26
itershards() (*DiskDataset method*), 34

J

j2c (*BaseDF property*), 712
j3c (*BaseDF property*), 712
jac() (*in module deepchem.utils.differentiation_utils.grad*), 768
jaccard_index() (*in module deepchem.metrics*), 601
jaccard_score() (*in module deepchem.metrics*), 590
Jacobian (*class in deepchem.utils.differentiation_utils.optimize.jacobian*), 785
JaxModel (*class in deepchem.models*), 428
JsonLoader (*class in deepchem.data*), 52

K

k_fold_split() (*ButinaSplitter method*), 239
k_fold_split() (*FingerprintSplitter method*), 242
k_fold_split() (*IndexSplitter method*), 222
k_fold_split() (*MaxMinSplitter method*), 236
k_fold_split() (*MolecularWeightSplitter method*), 233
k_fold_split() (*RandomGroupSplitter method*), 214
k_fold_split() (*RandomSplitter method*), 211
k_fold_split() (*RandomStratifiedSplitter method*), 217
k_fold_split() (*ScaffoldSplitter method*), 230
k_fold_split() (*SingletaskStratifiedSplitter method*), 219
k_fold_split() (*SpecifiedSplitter method*), 225
k_fold_split() (*Splitter method*), 244
k_fold_split() (*TaskSplitter method*), 227
kappa_score() (*in module deepchem.metrics*), 602
KerasModel (*class in deepchem.models*), 303
kpts (*BaseHamilton property*), 714

L

L1Loss (*class in deepchem.models.losses*), 288
L2Loss (*class in deepchem.models.losses*), 288
Lattice (*class in deepchem.utils.dft_utils*), 685
lattice_vectors() (*Lattice method*), 686
LCNNFeaturizer (*class in deepchem.feat*), 157
LCNNModel (*class in deepchem.models*), 381
LearningRateSchedule (*class in deepchem.models.optimizers*), 299
legacy_metadata (*DiskDataset attribute*), 31
LibXCGGA (*class in deepchem.utils.dft_utils.xc.libxc*), 707
LibXCLDA (*class in deepchem.utils.dft_utils.xc.libxc*), 705
LibXCMGGA (*class in deepchem.utils.dft_utils.xc.libxc*), 708
Linear (*class in deepchem.models.jax_models.layers*), 574

LinearCosineDecay (*class in deepchem.models.optimizers*), 302
LinearMixing (*class in deepchem.utils.differentiation_utils.optimize.jacobian*), 788
linearmixing() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 784
LinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 744
load() (*GroverAtomVocabularyBuilder class method*), 170, 172
load() (*VocabularyBuilder class method*), 208
load_bace_classification() (*in module deepchem.molnet*), 76
load_bace_regression() (*in module deepchem.molnet*), 76
load_bandgap() (*in module deepchem.molnet*), 90
load_bbbc001() (*in module deepchem.molnet*), 77
load_bbbc002() (*in module deepchem.molnet*), 77
load_bbbc003() (*in module deepchem.molnet*), 78
load_bbbc004() (*in module deepchem.molnet*), 79
load_bbbc005() (*in module deepchem.molnet*), 80
load_bbbp() (*in module deepchem.molnet*), 81
load_cell_counting() (*in module deepchem.molnet*), 82
load_chembl() (*in module deepchem.molnet*), 82
load_chembl25() (*in module deepchem.molnet*), 83
load_clearance() (*in module deepchem.molnet*), 83
load_clintox() (*in module deepchem.molnet*), 84
load_csv_files() (*in module deepchem.utils.data_utils*), 647
load_data() (*in module deepchem.utils.data_utils*), 646
load_dataset_from_disk() (*in module deepchem.utils.data_utils*), 648
load_delaney() (*in module deepchem.molnet*), 85
load_docked_ligands() (*in module deepchem.utils.docking_utils*), 678
load_factors() (*in module deepchem.molnet*), 85
load_freesolv() (*in module deepchem.molnet*), 86
load_from_disk() (*in module deepchem.utils.data_utils*), 648
load_from_pretrained() (*HuggingFaceModel method*), 437
load_from_pretrained() (*KerasModel method*), 310
load_from_pretrained() (*ModularTorchModel method*), 351
load_from_pretrained() (*TorchModel method*), 348
load_hiv() (*in module deepchem.molnet*), 87
load_hopv() (*in module deepchem.molnet*), 87
load_hppb() (*in module deepchem.molnet*), 88
load_json_files() (*in module deepchem.utils.data_utils*), 647
load_kaggle() (*in module deepchem.molnet*), 89
load_kinase() (*in module deepchem.molnet*), 89
load_lipo() (*in module deepchem.molnet*), 90
load_metadata() (*DiskDataset method*), 32
load_molecule() (*in module deepchem.utils.rdkit_utils*), 652
load_mp_formation_energy() (*in module deepchem.molnet*), 92
load_mp_metallicity() (*in module deepchem.molnet*), 94
load_muv() (*in module deepchem.molnet*), 95
load_nci() (*in module deepchem.molnet*), 96
load_pcba() (*in module deepchem.molnet*), 96
load_pdbbind() (*in module deepchem.molnet*), 97
load_perovskite() (*in module deepchem.molnet*), 91
load_pickle_files() (*in module deepchem.utils.data_utils*), 648
load_Platinum_Adsorption() (*in module deepchem.molnet*), 109
load_ppb() (*in module deepchem.molnet*), 98
load_qm7() (*in module deepchem.molnet*), 98
load_qm8() (*in module deepchem.molnet*), 100
load_qm9() (*in module deepchem.molnet*), 101
load_sampl() (*in module deepchem.molnet*), 102
load_sdf_files() (*in module deepchem.utils.data_utils*), 647
load_sider() (*in module deepchem.molnet*), 103
load_thermosol() (*in module deepchem.molnet*), 104
load_tox21() (*in module deepchem.molnet*), 104
load_toxcast() (*in module deepchem.molnet*), 105
load_uspto() (*in module deepchem.molnet*), 106
load_uv() (*in module deepchem.molnet*), 107
load_zinc15() (*in module deepchem.molnet*), 108
LocalGlobalDiscriminator (*class in deepchem.models.torch_models.gnn*), 513
LocalMutualInformationLoss (*class in deepchem.models.losses*), 291
log_prob_gaussian() (*ElectronSampler method*), 683
LogM3Transformation (*class in deepchem.utils.dft_utils.grid.radial_grid*), 729
LogTransformer (*class in deepchem.trans*), 253
Loss (*class in deepchem.models.losses*), 288
loss_func() (*GNNModular method*), 377
loss_func() (*GroverModel method*), 392
loss_func() (*InfoGraphModel method*), 372
loss_func() (*InfoGraphStarModel method*), 374
loss_func() (*InfoMax3DModular method*), 381
loss_func() (*ModularTorchModel method*), 350
LowRankMatrix (*class in deepchem.utils.differentiation_utils.optimize.jacobian*), 788
LSTMStep (*class in deepchem.models.layers*), 453
lsymeig() (*in module*

deepchem.utils.differentiation_utils.symeig), 765

M

m() (*LinearOperator class method*), 745
MACCSKeysFingerprint (*class in deepchem.feat*), 131
mae_score() (*in module deepchem.metrics*), 602
make_copy() (*BaseSystem method*), 724
make_pytorch_dataset() (*Dataset method*), 70
make_pytorch_dataset() (*DiskDataset method*), 35
make_pytorch_dataset() (*ImageDataset method*), 43
make_pytorch_dataset() (*NumpyDataset method*), 27
make_tf_dataset() (*Dataset method*), 70
make_tf_dataset() (*DiskDataset method*), 40
make_tf_dataset() (*ImageDataset method*), 45
make_tf_dataset() (*NumpyDataset method*), 29
MAML (*class in deepchem.metalearning*), 614
MAML (*class in deepchem.metalearning.torch_maml*), 615
map_unique_objs() (*Uniquifier method*), 811
mask_token (*RobertaFeaturizer property*), 191
mask_token_id (*RobertaFeaturizer property*), 191
masked_edge_loss_loader() (*GNNModular method*), 377
masked_node_loss_loader() (*GNNModular method*), 377
MaskedAffineFlow (*class in deepchem.models.torch_models.flows*), 559
match_dim() (*in module deepchem.utils.differentiation_utils*), 743
MATEmbedding (*class in deepchem.models.torch_models.layers*), 499
MATEncoderLayer (*class in deepchem.models.torch_models.layers*), 495
MaterialCompositionFeaturizer (*class in deepchem.feat*), 206
MaterialStructureFeaturizer (*class in deepchem.feat*), 207
MatExpOrbParams (*class in deepchem.utils.dft_utils*), 721
MATFeaturizer (*class in deepchem.feat*), 131
MATGenerator (*class in deepchem.models.torch_models.layers*), 500
MathWarning (*class in deepchem.utils.misc_utils*), 810
MATModel (*class in deepchem.models.torch_models*), 384
matmul() (*LinearOperator method*), 747
MatmulLinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 754
matrix_mul() (*IRVTransformer static method*), 272
MatrixLinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 755
matthews_corrcoef() (*in module deepchem.metrics*), 578
max_len_sentences_pair (*RobertaFeaturizer property*), 191
max_len_single_sentence (*RobertaFeaturizer property*), 191
MaxMinSplitter (*class in deepchem.splits*), 235
mean_absolute_error() (*in module deepchem.metrics*), 584
mean_squared_error() (*in module deepchem.metrics*), 583
MEGNetModel (*class in deepchem.models*), 383
memory_cache_size (*DiskDataset property*), 39
merge() (*DiskDataset static method*), 36
merge() (*NumpyDataset static method*), 28
merge_molecular_fragments() (*in module deepchem.utils.fragment_utils*), 655
merge_overlapping_boxes() (*in module deepchem.utils.coordinate_box_utils*), 658
message() (*MXMNetGlobalMessagePassing method*), 541
message_func() (*PNALayer method*), 517
message_function() (*Net3DLayer method*), 522
MessagePassing (*class in deepchem.models.layers*), 486
metadata_df (*DiskDataset attribute*), 31
MetaLearner (*class in deepchem.metalearning*), 613
Metric (*class in deepchem.metrics*), 605
minimize() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 793
minimize_conformers() (*ConformerGenerator method*), 651
MinMaxTransformer (*class in deepchem.trans*), 249
MLP_flow (*class in deepchem.models.torch_models.flows*), 562
mm() (*LinearOperator method*), 746
Model (*class in deepchem.models*), 283
ModularTorchModel (*class in deepchem.models.torch_models.modular*), 349
Mol2VecFingerprint (*class in deepchem.feat*), 135
MolecularFeaturizer (*class in deepchem.feat*), 206
MolecularFragment (*class in deepchem.utils.fragment_utils*), 653
MolecularWeightSplitter (*class in deepchem.splits*), 232
MoleculeLoadException (*class in deepchem.utils.rdkit_utils*), 651
MolGANAggregationLayer (*class in deepchem.models.layers*), 450
MolGANAggregationLayer (*class in deepchem.models.torch_models.layers*), 531
MolGANConvolutionLayer (*class in deepchem.models.layers*), 448
MolGANConvolutionLayer (*class in deepchem.models.torch_models.layers*), 529

MolGANEncoderLayer (*class in deepchem.models.layers*), 452
MolGANEncoderLayer (*class in deepchem.models.torch_models.layers*), 533
MolGanFeaturizer (*class in deepchem.feat*), 116
MolGANMultiConvolutionLayer (*class in deepchem.models.layers*), 451
MolGANMultiConvolutionLayer (*class in deepchem.models.torch_models.layers*), 532
MolGraphConvFeaturizer (*class in deepchem.feat*), 117
MordredDescriptors (*class in deepchem.feat*), 138
move() (*DiskDataset method*), 33
move() (*ElectronSampler method*), 684
MPNNModel (*class in deepchem.models*), 324
MPNNModel (*class in deepchem.models.torch_models*), 369
MSA_to_dataset() (*in module deepchem.utils.sequence_utils*), 664
MulBaseXC (*class in deepchem.utils.dft_utils.xc.base_xc*), 697
MulLinearOperator (*class in deepchem.utils.differentiation_utils.linop*), 752
MultiConvMol (*class in deepchem.feat.mol_graphs*), 63
MultiHeadedMATAttention (*class in deepchem.models.torch_models.layers*), 497
MultilayerPerceptron (*class in deepchem.models.torch_models.layers*), 492
MultitaskClassifier (*class in deepchem.models*), 358
MultitaskFitTransformRegressor (*class in deepchem.models*), 356
MultitaskRegressor (*class in deepchem.models*), 355
mv() (*FullRankMatrix method*), 790
mv() (*LinearOperator method*), 746
mv() (*LowRankMatrix method*), 789
MXMNetBesselBasisLayer (*class in deepchem.models.torch_models.layers*), 541
MXMNetFeaturizer (*class in deepchem.feat*), 124
MXMNetGlobalMessagePassing (*class in deepchem.models.torch_models.layers*), 539
MXMNetLocalMessagePassing (*class in deepchem.models.torch_models.layers*), 551
MXMNetSphericalBasisLayer (*class in deepchem.models.torch_models.layers*), 554

N

n_actions (*Environment property*), 619
nao (*BaseHamilton property*), 714
NeighborList (*class in deepchem.models.layers*), 463
Net3D (*class in deepchem.models.torch_models.gnn3d*), 522
Net3DLayer (*class in deepchem.models.torch_models.gnn3d*), 521
node_features (*GraphData attribute*), 64
node_pos_features (*GraphData attribute*), 64
nonlinearity() (*VinaFreeEnergy method*), 462
NormalizationTransformer (*class in deepchem.trans*), 247
normalize() (*CoulombFitTransformer method*), 269
normalize_bcast_dims() (*in module deepchem.utils.differentiation_utils*), 743
normalize_labels_shape() (*in module deepchem.metrics*), 577
normalize_prediction_shape() (*in module deepchem.metrics*), 577
normalize_weight_shape() (*in module deepchem.metrics*), 576
NormalizingFlow (*class in deepchem.models.torch_models*), 386
NormalizingFlowModel (*class in deepchem.models.normalizing_flows*), 340
NTXentMultiplePositives (*class in deepchem.models.losses*), 298
num_edges (*GraphData attribute*), 64
num_edges_features (*GraphData attribute*), 65
num_node_features (*GraphData attribute*), 64
num_nodes (*GraphData attribute*), 64
num_special_tokens_to_add() (*RobertaFeaturizer method*), 192
numel (*BaseSystem property*), 725
numpy_to_torch() (*GraphData method*), 66
NumpyDataset (*class in deepchem.data*), 24

O

objparams() (*PureFunction method*), 756, 759
occnumber() (*in module deepchem.utils.safeops_utils*), 812
one_hot_encode() (*in module deepchem.utils.molecule_feature_utils*), 672
one_of_k_encoding() (*in module deepchem.feat.graph_features*), 125
one_of_k_encoding_unk() (*in module deepchem.feat.graph_features*), 126
OneHotFeaturizer (*class in deepchem.feat*), 146
Optimizer (*class in deepchem.models.optimizers*), 299
orb2params() (*BaseOrbParams static method*), 720
orb2params() (*MatExpOrbParams static method*), 722
orb2params() (*QROrbParams static method*), 721
ortho() (*in module deepchem.utils.differentiation_utils.symeig*), 768
output_node_func() (*Net3D method*), 524
output_predictions() (*Evaluator method*), 660
output_statistics() (*Evaluator method*), 660

P

pack_aoparams() (*HFEngine method*), 736
pad() (*RobertaFeaturizer method*), 192

pad_array() (*in module deepchem.utils.data_utils*), 645
pad_array() (*MATModel method*), 385
pad_sequence() (*MATModel method*), 386
pad_smile() (*OneHotFeaturizer method*), 147
pad_string() (*OneHotFeaturizer method*), 147
pad_token (*RobertaFeaturizer property*), 193
pad_token_id (*RobertaFeaturizer property*), 193
pad_token_type_id (*RobertaFeaturizer property*), 193
PagtnModel (*class in deepchem.models*), 365
PagtnMolGraphFeaturizer (*class in deepchem.feat*), 119
pair_features() (*in module deepchem.feat.graph_features*), 130
pairwise_distances() (*in module deepchem.dock.pose_scoring*), 643
parameters() (*MetaLearner method*), 614
params (*Lattice property*), 686
params2orb() (*BaseOrbParams static method*), 720
params2orb() (*MatExpOrbParams static method*), 721
params2orb() (*QROrbParams static method*), 720
parse_moldesc (*class in deepchem.utils.dft_utils.api.parser*), 722
pearson_r2_score() (*in module deepchem.metrics*), 601
PFMFeaturizer (*class in deepchem.feat.sequence_featurizers*), 173
PINNModel (*class in deepchem.models*), 432
pixel_error() (*in module deepchem.metrics*), 601
PNA (*class in deepchem.models.torch_models*), 519
PNAGNN (*class in deepchem.models.torch_models.pna_gnn*), 518
PNALayer (*class in deepchem.models.torch_models.pna_gnn*), 516
PoissonLoss (*class in deepchem.models.losses*), 288
polarized (*HFEngine property*), 735
Policy (*class in deepchem.rl*), 621
PolynomialDecay (*class in deepchem.models.optimizers*), 301
PoseGenerator (*class in deepchem.dock.pose_generation*), 638
PositionwiseFeedForward (*class in deepchem.models.torch_models.layers*), 498
possible_atom_list (*GraphConvConstants attribute*), 125
possible_bond_stereo (*GraphConvConstants attribute*), 125
possible_chirality_list (*GraphConvConstants attribute*), 125
possible_formal_charge_list (*GraphConvConstants attribute*), 125
possible_hybridization_list (*GraphConvConstants attribute*), 125
possible_number_radical_e_list (*GraphConvConstants attribute*), 125
possible_numH_list (*GraphConvConstants attribute*), 125
possible_valence_list (*GraphConvConstants attribute*), 125
PowerTransformer (*class in deepchem.trans*), 257
PPO (*class in deepchem.rl.ppo*), 624
PPO (*class in deepchem.rl.torch_rl*), 632
PPOLoss (*class in deepchem.rl.ppo*), 626
PPOLoss (*class in deepchem.rl.torch_rl*), 636
prc_auc_score() (*in module deepchem.metrics*), 602
precision_recall_curve() (*in module deepchem.metrics*), 588
precision_score() (*in module deepchem.metrics*), 585
predict() (*A2C method*), 623, 631
predict() (*JaxModel method*), 431
predict() (*KerasModel method*), 308
predict() (*Model method*), 284
predict() (*PPO method*), 626, 635
predict() (*SklearnModel method*), 286
predict() (*TorchModel method*), 345
predict_embedding() (*KerasModel method*), 308
predict_embedding() (*SeqToSeqModel method*), 398
predict_embedding() (*TorchModel method*), 346
predict_embeddings() (*SeqToSeq method*), 330
predict_from_embedding() (*SeqToSeqModel method*), 398
predict_from_embeddings() (*SeqToSeq method*), 330
predict_from_sequences() (*SeqToSeq method*), 330
predict_from_sequences() (*SeqToSeqModel method*), 398
predict_gan_generator() (*BasicMolGANModel method*), 327, 413
predict_gan_generator() (*GAN method*), 333
predict_gan_generator() (*GANModel method*), 407
predict_on_batch() (*JaxModel method*), 430
predict_on_batch() (*KerasModel method*), 307
predict_on_batch() (*MAML method*), 615, 618
predict_on_batch() (*Model method*), 283
predict_on_batch() (*SklearnModel method*), 286
predict_on_batch() (*TorchModel method*), 345
predict_on_generator() (*JaxModel method*), 430
predict_on_generator() (*KerasModel method*), 306
predict_on_generator() (*MultitaskFitTransformRegressor method*), 357
predict_on_generator() (*TorchModel method*), 345
predict_uncertainty() (*KerasModel method*), 308
predict_uncertainty() (*TorchModel method*), 346
predict_uncertainty_on_batch() (*KerasModel method*), 307
predict_uncertainty_on_batch() (*TorchModel method*), 345
prepare_for_model() (*RobertaFeaturizer method*), 193

prepare_inputs() (*in module deepchem.utils.docking_utils*), 678
prepare_seq2seq_batch() (*RobertaFeaturizer method*), 196
pretrans_edges() (*PNALayer method*), 518
ProgressiveMultitaskClassifier (*class in deepchem.models*), 314
ProgressiveMultitaskModel (*class in deepchem.models.torch_models*), 419
ProgressiveMultitaskRegressor (*class in deepchem.models*), 315
prune_conformers() (*ConformerGenerator method*), 651
PubChemFingerprint (*class in deepchem.feat*), 134
PureFunction (*class in deepchem.utils.differentiation_utils.pure_function*), 756, 758
push_to_hub() (*RobertaFeaturizer method*), 197

Q

QROrbParams (*class in deepchem.utils.dft_utils*), 720

R

r2_score() (*in module deepchem.metrics*), 581
radial_cutoff() (*AtomicConvolution method*), 467
radial_symmetry() (*ANIFeat method*), 472
radial_symmetry_function() (*AtomicConvolution method*), 466
RadialGrid (*class in deepchem.utils.dft_utils.grid.radial_grid*), 725
RandomGroupSplitter (*class in deepchem.splits*), 213
randomize_coulomb_matrix() (*CoulombMatrix method*), 140
randomize_coulomb_matrix() (*CoulombMatrixEig method*), 142
RandomSplitter (*class in deepchem.splits*), 210
RandomStratifiedSplitter (*class in deepchem.splits*), 215
RawFeaturizer (*class in deepchem.feat*), 149
RDKitConformerFeaturizer (*class in deepchem.feat*), 123
RDKitDescriptors (*class in deepchem.feat*), 136
RdkitGridFeaturizer (*class in deepchem.feat*), 150
read_gnina_log() (*in module deepchem.utils.docking_utils*), 679
realize() (*CoulombFitTransformer method*), 269
realize() (*IRVTransformer method*), 271
RealNVPLayer (*class in deepchem.models.torch_models.layers*), 504
recall_score() (*in module deepchem.metrics*), 579
recip_vectors() (*Lattice method*), 686
reduce() (*FullRankMatrix method*), 791
reduce() (*LowRankMatrix method*), 790
reduce() (*SpinParam method*), 687
reduce_func() (*PNALayer method*), 517
reduce_molecular_complex_to_contacts() (*in module deepchem.utils.fragment_utils*), 656
reference_lists (*GraphConvConstants attribute*), 125
register_for_auto_class() (*RobertaFeaturizer class method*), 198
relative_difference() (*in module deepchem.utils.evaluate*), 662
reload() (*AtomicConvModel method*), 337
reload() (*Model method*), 283
reload() (*NormalizingFlowModel method*), 341
reload() (*SklearnModel method*), 286
remove_pad() (*SmilesToSeq method*), 145
repulsion() (*VinaFreeEnergy method*), 462
requires_grid() (*BaseSystem method*), 724
reset() (*Environment method*), 620
reset() (*GymEnvironment method*), 620
reset_parameters() (*AtomEncoder method*), 515
reset_parameters() (*MXMNetBesselBasisLayer method*), 542
reshard() (*DiskDataset method*), 33
restore() (*A2C method*), 624, 632
restore() (*GroverModel method*), 393
restore() (*InfoGraphModel method*), 372
restore() (*KerasModel method*), 310
restore() (*MAML method*), 615, 617
restore() (*ModularTorchModel method*), 352
restore() (*PPO method*), 626, 636
restore() (*TorchModel method*), 348
restore_objparams() (*PureFunction method*), 757, 759
rmm() (*LinearOperator method*), 747
rms_score() (*in module deepchem.metrics*), 602
RMSProp (*class in deepchem.models.optimizers*), 301
rmv() (*FullRankMatrix method*), 790
rmv() (*LinearOperator method*), 747
rmv() (*LowRankMatrix method*), 789
RobertaFeaturizer (*class in deepchem.feat*), 175
RobustMultitaskClassifier (*class in deepchem.models*), 311
RobustMultitaskRegressor (*class in deepchem.models*), 313
roc_auc_score() (*in module deepchem.metrics*), 594
rootfinder() (*in module deepchem.utils.differentiation_utils.optimize.rootfinder*), 791
run() (*BaseQCCalc method*), 738
run() (*SCF_QCCalc method*), 739
RxnFeaturizer (*class in deepchem.feat*), 202
RxnSplitTransformer (*class in deepchem.trans*), 275

S

safe_cdist() (*in module deepchem.utils.safeops_utils*), 813

safe_index() (*in module deepchem.feat.graph_features*), 126
safedenom() (*in module deepchem.utils.differentiation_utils.solve*), 775
safenorm() (*in module deepchem.utils.safeops_utils*), 812
safepow() (*in module deepchem.utils.safeops_utils*), 811
SAMFeaturizer (*class in deepchem.feat*), 160
SAMLoader (*class in deepchem.data.data_loader*), 59
sample() (*FakeGraphGenerator method*), 681
sampled_electrons (*ElectronSampler attribute*), 683
sanitize_special_tokens() (*RobertaFeaturizer method*), 198
save() (*AtomicConvModel method*), 337
save() (*GroverAtomVocabularyBuilder method*), 169, 172
save() (*Model method*), 284
save() (*NormalizingFlowModel method*), 341
save() (*SklearnModel method*), 286
save() (*VocabularyBuilder method*), 208
save_checkpoint() (*A2C method*), 632
save_checkpoint() (*KerasModel method*), 310
save_checkpoint() (*MAML method*), 618
save_checkpoint() (*ModularTorchModel method*), 352
save_checkpoint() (*PPO method*), 636
save_checkpoint() (*TorchModel method*), 347
save_dataset_to_disk() (*in module deepchem.utils.data_utils*), 649
save_pretrained() (*RobertaFeaturizer method*), 198
save_to_disk() (*DiskDataset method*), 32
save_to_disk() (*in module deepchem.utils.data_utils*), 648
save_vocabulary() (*RobertaFeaturizer method*), 199
save_vocabulary() (*SmilesTokenizer method*), 165
ScaffoldSplitter (*class in deepchem.splits*), 228
scale_amplification() (*in module deepchem.utils.graph_utils*), 669
scale_attenuation() (*in module deepchem.utils.graph_utils*), 669
scale_identity() (*in module deepchem.utils.graph_utils*), 668
ScaledDotProductAttention (*class in deepchem.models.torch_models.attention*), 572
ScaleNorm (*class in deepchem.models.torch_models.layers*), 494
SCF_QCCalc (*class in deepchem.utils.dft_utils.qccalc.scf_qccalc*), 739
scipy_linalg_op() (*LinearOperator method*), 747
scp2dm() (*HFEngine method*), 735
scp2scp() (*HFEngine method*), 735
ScScoreModel (*class in deepchem.models*), 327
SDFLoader (*class in deepchem.data*), 53
segment_sum() (*in module deepchem.utils.pytorch_utils*), 797
select() (*Dataset method*), 69
select() (*DiskDataset method*), 39
select() (*ImageDataset method*), 43
select() (*NumpyDataset method*), 26
select_action() (*A2C method*), 623, 631
select_action() (*PPO method*), 626, 635
select_task() (*MetaLearner method*), 614
SelfAttention (*class in deepchem.models.torch_models.attention*), 573
sentences2vec() (*Mol2VecFingerprint method*), 136
sep_token (*RobertaFeaturizer property*), 199
sep_token_id (*RobertaFeaturizer property*), 199
seq_one_hot_encode() (*in module deepchem.utils.genomics_utils*), 662
SeqToSeq (*class in deepchem.models*), 328
SeqToSeq (*class in deepchem.models.torch_models.seqtoseq*), 546
SeqToSeqModel (*class in deepchem.models.torch_models*), 396
set_attr (*class in deepchem.utils.attribute_utils*), 796
set_cache() (*BaseSystem method*), 723
set_default_option() (*in module deepchem.utils.differentiation_utils*), 760
set_eigen_options() (*HFEngine method*), 736
set_max_print_size() (*in module deepchem.utils.debug_utils*), 680
set_objparams() (*PureFunction method*), 756, 759
set_print_threshold() (*in module deepchem.utils.debug_utils*), 680
set_shard() (*DiskDataset method*), 38
set_truncation_and_padding() (*RobertaFeaturizer method*), 200
SetGather (*class in deepchem.models.layers*), 489
SetGather (*class in deepchem.models.torch_models.layers*), 511
setparams() (*EditableModule method*), 741
setuniqueparams() (*EditableModule method*), 742
setup() (*BroydenFirst method*), 787
setup() (*Jacobian method*), 786
setup() (*LinearMixing method*), 788
setup_grid() (*BaseHamilton method*), 714
setup_grid() (*BaseSystem method*), 724
setup_linear_problem() (*in module deepchem.utils.differentiation_utils.solve*), 774
setup_precond() (*in module deepchem.utils.differentiation_utils.solve*), 773
ShannonEntropy (*class in deepchem.models.losses*), 290
shape (*HFEngine property*), 734

shape (*LinearOperator property*), 750
shape2str() (*in module deepchem.utils.misc_utils*), 810
shuffle_each_shard() (*DiskDataset method*), 37
shuffle_shards() (*DiskDataset method*), 37
SigmoidCrossEntropy (*class in deepchem.models.losses*), 288
SineCoulombMatrix (*class in deepchem.feat*), 155
SingletaskStratifiedSplitter (*class in deepchem.splits*), 219
SklearnModel (*class in deepchem.models*), 285
SlicedRadialGrid (*class in deepchem.utils.dft_utils.grid.radial_grid*), 728
slow_tokenizer_class (*RobertaFeaturizer attribute*), 200
SluiceLoss (*class in deepchem.models.layers*), 469
Smiles2Vec (*class in deepchem.models*), 337
smiles_from_seq() (*SmilesToSeq method*), 145
smiles_to_seq() (*TextCNNModel method*), 335, 423
smiles_to_seq_batch() (*TextCNNModel method*), 335, 424
SmilesToImage (*class in deepchem.feat*), 145
SmilesTokenizer (*class in deepchem.feat*), 163
SmilesToSeq (*class in deepchem.feat*), 144
SNAPFeaturizer (*class in deepchem.feat*), 149
so3_generators() (*in module deepchem.utils.equivariance_utils*), 807
SoftmaxCrossEntropy (*class in deepchem.models.losses*), 288
solve() (*BroydenFirst method*), 787
solve() (*in module deepchem.utils.differentiation_utils.solve*), 776
solve() (*Jacobian method*), 786
solve() (*LinearMixing method*), 788
solve_ABE() (*in module deepchem.utils.differentiation_utils.solve*), 772
solve_torchfcn (*class in deepchem.utils.differentiation_utils.solve*), 779
sparse_shuffle() (*DiskDataset method*), 37
SparseAdam (*class in deepchem.models.optimizers*), 300
SparseMatrixOneHotFeaturizer (*class in deepchem.feat*), 148
SparseSoftmaxCrossEntropy (*class in deepchem.models.losses*), 288
special_tokens_map (*RobertaFeaturizer property*), 200
special_tokens_map_extended (*RobertaFeaturizer property*), 200
SpecifiedSplitter (*class in deepchem.splits*), 224
spin (*BaseSystem property*), 725
SpinParam (*class in deepchem.utils.dft_utils*), 687
split() (*ButinaSplitter method*), 238
split() (*FingerprintSplitter method*), 241
split() (*IndexSplitter method*), 221
split() (*MaxMinSplitter method*), 235
split() (*MolecularWeightSplitter method*), 232
split() (*RandomGroupSplitter method*), 213
split() (*RandomSplitter method*), 210
split() (*RandomStratifiedSplitter method*), 216
split() (*ScaffoldSplitter method*), 229
split() (*SingletaskStratifiedSplitter method*), 219
split() (*SpecifiedSplitter method*), 225
split() (*Splitter method*), 245
split() (*TaskSplitter method*), 227
Splitter (*class in deepchem.splits*), 244
SquaredHingeLoss (*class in deepchem.models.losses*), 288
Stack (*class in deepchem.models.layers*), 460
state (*Environment property*), 619
state_dtype (*Environment property*), 619
state_shape (*Environment property*), 619
step() (*Environment method*), 620
step() (*GymEnvironment method*), 620
strip_hydrogens() (*in module deepchem.utils.fragment_utils*), 655
su2_generators() (*in module deepchem.utils.equivariance_utils*), 806
subgraph() (*GraphData method*), 66
SublayerConnection (*class in deepchem.models.torch_models.layers*), 498
subset() (*DiskDataset method*), 36
sum() (*SpinParam method*), 687
sum_neigh() (*GraphConv method*), 446
SwitchedDropout (*class in deepchem.models.layers*), 456
symeig() (*in module deepchem.utils.differentiation_utils.symeig*), 765
symeig_torchfcn (*class in deepchem.utils.differentiation_utils.symeig*), 767

T

tallqr() (*in module deepchem.utils.pytorch_utils*), 799
TaskSplitter (*class in deepchem.splits*), 227
TensorflowMultitaskIRVClassifier (*class in deepchem.models*), 311
TensorNonTensorSeparator() (*in module deepchem.utils.pytorch_utils*), 799
terminated (*Environment property*), 619
TerminationCondition() (*in module deepchem.utils.differentiation_utils.optimize.minimizer*), 782
TerminationCondition() (*in module deepchem.utils.differentiation_utils.optimize.rootsolver*), 785
TextCNNModel (*class in deepchem.models*), 334

TextCNNModel (*class in deepchem.models.torch_models*), 422
THRESHOLD_MEMORY (*_Config attribute*), 719
to_csv() (*Dataset method*), 71
to_csv() (*DiskDataset method*), 41
to_csv() (*ImageDataset method*), 45
to_csv() (*NumpyDataset method*), 29
to_dataframe() (*Dataset method*), 70
to_dataframe() (*DiskDataset method*), 41
to_dataframe() (*ImageDataset method*), 45
to_dataframe() (*NumpyDataset method*), 29
to_dgl_graph() (*GraphData method*), 65
to_fortran_order() (*in module deepchem.utils.pytorch_utils*), 800
to_json() (*NumpyDataset method*), 27
to_one_hot() (*in module deepchem.metrics*), 575
to_pyg_graph() (*GraphData method*), 65
to_seq() (*SmilesToSeq method*), 145
tokenize() (*BasicSmilesTokenizer method*), 166
tokenize() (*RobertaFeaturizer method*), 200
top_k_accuracy_score() (*in module deepchem.metrics*), 600
TorchModel (*class in deepchem.models*), 341
TorchNNPureFunction (*class in deepchem.utils.differentiation_utils.pure_function*), 758
train_new_from_iterator() (*RobertaFeaturizer method*), 201
train_on_current_task() (*MAML method*), 615, 618
train_test_split() (*ButinaSplitter method*), 239
train_test_split() (*FingerprintSplitter method*), 242
train_test_split() (*IndexSplitter method*), 223
train_test_split() (*MaxMinSplitter method*), 237
train_test_split() (*MolecularWeightSplitter method*), 234
train_test_split() (*RandomGroupSplitter method*), 214
train_test_split() (*RandomSplitter method*), 211
train_test_split() (*RandomStratifiedSplitter method*), 217
train_test_split() (*ScaffoldSplitter method*), 231
train_test_split() (*SingletaskStratifiedSplitter method*), 220
train_test_split() (*SpecifiedSplitter method*), 225
train_test_split() (*Splitter method*), 245
train_test_split() (*TaskSplitter method*), 228
train_valid_test_split() (*ButinaSplitter method*), 240
train_valid_test_split() (*FingerprintSplitter method*), 243
train_valid_test_split() (*IndexSplitter method*), 223
train_valid_test_split() (*MaxMinSplitter method*), 237
train_valid_test_split() (*MolecularWeightSplitter method*), 234
train_valid_test_split() (*RandomGroupSplitter method*), 214
train_valid_test_split() (*RandomSplitter method*), 212
train_valid_test_split() (*RandomStratifiedSplitter method*), 218
train_valid_test_split() (*ScaffoldSplitter method*), 231
train_valid_test_split() (*SingletaskStratifiedSplitter method*), 220
train_valid_test_split() (*SpecifiedSplitter method*), 226
train_valid_test_split() (*Splitter method*), 244
train_valid_test_split() (*TaskSplitter method*), 227
training_step() (*DCLightningModule method*), 427
transform() (*BalancingTransformer method*), 261
transform() (*CDFTransformer method*), 257
transform() (*ClippingTransformer method*), 252
transform() (*CoulombFitTransformer method*), 270
transform() (*DAGTransformer method*), 274
transform() (*Dataset method*), 69
transform() (*DiskDataset method*), 35
transform() (*DuplicateBalancingTransformer method*), 264
transform() (*FeaturizationTransformer method*), 267
transform() (*ImageDataset method*), 43
transform() (*ImageTransformer method*), 265
transform() (*IRVTransformer method*), 272
transform() (*LogTransformer method*), 254
transform() (*MinMaxTransformer method*), 250
transform() (*NormalizationTransformer method*), 248
transform() (*NumpyDataset method*), 26
transform() (*PowerTransformer method*), 259
transform() (*RxnSplitTransformer method*), 277
transform() (*Transformer method*), 278
transform_array() (*BalancingTransformer method*), 261
transform_array() (*CDFTransformer method*), 256
transform_array() (*ClippingTransformer method*), 252
transform_array() (*CoulombFitTransformer method*), 270
transform_array() (*DAGTransformer method*), 274
transform_array() (*DuplicateBalancingTransformer method*), 263
transform_array() (*FeaturizationTransformer method*), 267
transform_array() (*ImageTransformer method*), 265
transform_array() (*IRVTransformer method*), 272
transform_array() (*LogTransformer method*), 254

transform_array() (*MinMaxTransformer method*), 250
transform_array() (*NormalizationTransformer method*), 247
transform_array() (*PowerTransformer method*), 258
transform_array() (*RxnSplitTransformer method*), 276
transform_array() (*Transformer method*), 278
transform_on_array() (*BalancingTransformer method*), 261
transform_on_array() (*CDFTransformer method*), 257
transform_on_array() (*ClippingTransformer method*), 252
transform_on_array() (*CoulombFitTransformer method*), 270
transform_on_array() (*DAGTransformer method*), 275
transform_on_array() (*DuplicateBalancingTransformer method*), 264
transform_on_array() (*FeaturizationTransformer method*), 268
transform_on_array() (*ImageTransformer method*), 266
transform_on_array() (*IRVTransformer method*), 273
transform_on_array() (*LogTransformer method*), 255
transform_on_array() (*MinMaxTransformer method*), 250
transform_on_array() (*NormalizationTransformer method*), 248
transform_on_array() (*PowerTransformer method*), 259
transform_on_array() (*RxnSplitTransformer method*), 277
transform_on_array() (*Transformer method*), 279
Transformer (*class in deepchem.trans*), 278
TreutlerM4Transformation (*class in deepchem.utils.dft_utils.grid.radial_grid*), 730
truncate_sequences() (*RobertaFeaturizer method*), 201

U

UG_to_DAG() (*DAGTransformer method*), 274
UNetModel (*class in deepchem.models.torch_models*), 424
unfreeze_components() (*ModularTorchModel method*), 350
UnimplementedError (*class in deepchem.utils.misc_utils*), 810
union() (*in module deepchem.utils.coordinate_box_utils*), 658
Uniquifier (*class in deepchem.utils.misc_utils*), 811
unit_vector() (*in module deepchem.utils.geometry_utils*), 665
unk_token (*RobertaFeaturizer property*), 202
unk_token_id (*RobertaFeaturizer property*), 202
unpack_aoparams() (*HFEngine method*), 736
unsorted_segment_sum() (*in module deepchem.utils.pytorch_utils*), 797
untargz_file() (*in module deepchem.utils.data_utils*), 646
untransform() (*BalancingTransformer method*), 262
untransform() (*CDFTransformer method*), 256
untransform() (*ClippingTransformer method*), 252
untransform() (*CoulombFitTransformer method*), 270
untransform() (*DAGTransformer method*), 274
untransform() (*DuplicateBalancingTransformer method*), 265
untransform() (*FeaturizationTransformer method*), 268
untransform() (*ImageTransformer method*), 266
untransform() (*IRVTransformer method*), 272
untransform() (*LogTransformer method*), 254
untransform() (*MinMaxTransformer method*), 250
untransform() (*NormalizationTransformer method*), 248
untransform() (*OneHotFeaturizer method*), 148
untransform() (*PowerTransformer method*), 259
untransform() (*RxnSplitTransformer method*), 277
untransform() (*SparseMatrixOneHotFeaturizer method*), 148
untransform() (*Transformer method*), 279
untransform_grad() (*NormalizationTransformer method*), 248
unzip_file() (*in module deepchem.utils.data_utils*), 646
update() (*BroydenFirst method*), 787
update() (*Jacobian method*), 786
update() (*LinearMixing method*), 788
update_function() (*Net3DLayer method*), 522
Usage (*_Config attribute*), 719
uselinopparams() (*LinearOperator method*), 746
useobjparams() (*PureFunction method*), 757, 759
UserCSVLoader (*class in deepchem.data*), 48
UserDefinedFeaturizer (*class in deepchem.feat*), 204
usymeig() (*in module deepchem.utils.differentiation_utils.symeig*), 765

V

VAE_ELBO (*class in deepchem.models.losses*), 289
VAE_KLDivergence (*class in deepchem.models.losses*), 289
ValGrad (*class in deepchem.utils.dft_utils*), 687
variables (*MetaLearner property*), 614

VariationalRandomizer (*class in deepchem.models.torch_models.layers*), 543
vectorize() (*in module deepchem.utils.hash_utils*), 670
VERBOSE (*_Config attribute*), 719
vina_energy_term() (*in module deepchem.dock.pose_scoring*), 645
vina_gaussian_first() (*in module deepchem.dock.pose_scoring*), 644
vina_gaussian_second() (*in module deepchem.dock.pose_scoring*), 644
vina_hbond() (*in module deepchem.dock.pose_scoring*), 644
vina_hydrophobic() (*in module deepchem.dock.pose_scoring*), 643
vina_nonlinearity() (*in module deepchem.dock.pose_scoring*), 643
vina_repulsion() (*in module deepchem.dock.pose_scoring*), 643
VinaFreeEnergy (*class in deepchem.models.layers*), 461
VinaPoseGenerator (*class in deepchem.dock.pose_generation*), 639
vocab_size (*RobertaFeaturizer property*), 202
vocab_size (*SmilesTokenizer property*), 164
VocabularyBuilder (*class in deepchem.feat.vocabulary_builders.vocabulary_builder*), 208
volume() (*CoordinateBox method*), 657
volume() (*Lattice method*), 686
voxelize() (*in module deepchem.utils.voxel_utils*), 671

W

w (*Dataset property*), 68
w (*DiskDataset property*), 39
w (*ImageDataset property*), 42
w (*NumpyDataset property*), 25
Weave (*class in deepchem.models.torch_models*), 414
WeaveFeaturizer (*class in deepchem.feat*), 114
WeaveGather (*class in deepchem.models.layers*), 479
WeaveGather (*class in deepchem.models.torch_models.layers*), 537
WeaveLayer (*class in deepchem.models.layers*), 476
WeaveLayer (*class in deepchem.models.torch_models.layers*), 535
WeaveModel (*class in deepchem.models*), 317
WeaveModel (*class in deepchem.models.torch_models*), 416
WeaveMol (*class in deepchem.feat.mol_graphs*), 63
WeightedLinearCombo (*class in deepchem.models.layers*), 457
wfnormalize_() (*CGTOBasis method*), 689
WGAN (*class in deepchem.models*), 333
WGANModel (*class in deepchem.models.torch_models*), 408
wigner_D() (*in module deepchem.utils.equivariance_utils*), 809
wrap_gmres() (*in module deepchem.utils.differentiation_utils.solve*), 770
write_data_to_disk() (*DiskDataset static method*), 32
write_gnina_conf() (*in module deepchem.utils.docking_utils*), 678
write_molecule() (*in module deepchem.utils.rdkit_utils*), 652
write_vina_conf() (*in module deepchem.utils.docking_utils*), 677

X

X (*Dataset property*), 67
X (*DiskDataset property*), 39
X (*ImageDataset property*), 42
X (*NumpyDataset property*), 25
x2r() (*BaseGridTransform method*), 728
x2r() (*DE2Transformation method*), 729
x2r() (*LogM3Transformation method*), 730
x2r() (*TreutlerM4Transformation method*), 731
X_transform() (*CoulombFitTransformer method*), 269
X_transform() (*IRVTransformer method*), 272

Y

y (*Dataset property*), 68
y (*DiskDataset property*), 39
y (*ImageDataset property*), 42
y (*NumpyDataset property*), 25