deepchem

Release 2.6.0.dev

deepchem-contributors

Jan 11, 2022

GET STARTED

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 or this forum post for Colab quick start guides.

If you'd like to install DeepChem locally, we recommend installing deepchem which is nightly version and RDKit. RDKit is a soft requirement package, but many useful methods depend on it.

```
pip install tensorflow~=2.4
pip install --pre deepchem
conda install -y -c conda-forge rdkit
```

Then open your python and try running.

```
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

Please install tensorflow ~2.4 before installing deepchem.

```
pip install tensorflow~=2.4
```

Then, you install deepchem via pip or conda.

```
pip install deepchem
```

or

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

RDKit is a soft requirement package, but many useful methods like molnet depend on it. We recommend installing RDKit with deepchem if you use conda.

```
conda install -y -c conda-forge rdkit
```

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 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]'`

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 3.6.10 |Anaconda, Inc.| (default, May  8 2020, 02:54:21)
[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 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 3.7 cpu
```

If you want GPU support (we supports only CUDA 10.1):

```
source scripts/install_deepchem_conda.sh 3.7 gpu
```

If you are using the Windows and the PowerShell:

```
. .\scripts\install_deepchem_conda.ps1 3.7 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
```

3.1.6 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=3.8
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 would a install all the soft-dependencies at once, then take a look `deepchem/requirements/<https://github.com/deepchem/deepchem/tree/master/requirements>`___

3.2 Requirements

3.2.1 Hard requirements

DeepChem officially supports Python 3.6 through 3.7 and requires these packages on any condition.

- joblib
- NumPy
- pandas
- scikit-learn
- SciPy
- TensorFlow
  - *deepchem>=2.4.0* depends on TensorFlow v2 (2.3.x)
  - *deepchem<2.4.0* depends on TensorFlow v1 (>=1.14)

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`
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	1.6.0	`dc.data.datasets`
PyTorch Geometric	1.6.x (with PyTorch 1.6.0)	`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 Probability	0.11.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`

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

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',
```

(continues on next page)

(continued from previous page)

```
...   '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` conteins 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

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
>>>
>>> # Run before every test for reproducibility
>>> def seed_all():
...     np.random.seed(123)
...     tf.random.set_seed(123)
```

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,
```

(continues on next page)

(continued from previous page)

```
...     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

```
>>> # 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 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

3.6.1 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], y: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, ids: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], y: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, ids: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = 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() → numpy.ndarray

Get the names of the tasks associated with this dataset.

property X

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

property y

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

property ids

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

property w

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

iterbatches(*batch_size: Optional[int] = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: transformers.Transformer*, ***args*) → deepchem.data.datasets.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: Union[Sequence[int], numpy.ndarray]*, *select_dir: Optional[str] = None*) → deepchem.data.datasets.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: Optional[int] = 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: deepchem.data.datasets.DiskDataset*) → deepchem.data.datasets.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*) → deepchem.data.datasets.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[deepchem.data.datasets.Dataset]*) → deepchem.data.datasets.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: pandas.core.frame.DataFrame, X: Optional[Union[str, Sequence[str]]] = None, y: Optional[Union[str, Sequence[str]]] = None, w: Optional[Union[str, Sequence[str]]] = None, ids: Optional[str] = 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[numpy.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_dataframe() → pandas.core.frame.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]]*, *data_dir: Optional[str] = None*, *tasks: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtyp bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = []*) → deepchem.data.datasets.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], pandas.core.frame.DataFrame]

Helper method that loads metadata from disk.

static write_data_to_disk(*data_dir: str*, *basename: str*, *X: Optional[numpy.ndarray] = None*, *y: Optional[numpy.ndarray] = None*, *w: Optional[numpy.ndarray] = None*, *ids: Optional[numpy.ndarray] = 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: Optional[bool] = 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*) → deepchem.data.datasets.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() → numpy.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[int] = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: transformers.Transformer*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***args*) → deepchem.data.datasets.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: Optional[int] = 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]] bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], y: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]] bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]] bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, ids: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]] bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, tasks: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]] bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, data_dir: Optional[str] = None) → deepchem.data.datasets.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[deepchem.data.datasets.Dataset]*, *merge_dir: Optional[str] = None*) → deepchem.data.datasets.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: Optional[str] = None*) → deepchem.data.datasets.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: Optional[str] = None*) → deepchem.data.datasets.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: Optional[List[str]] = 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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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*) → numpy.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*) → numpy.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*) → numpy.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: numpy.ndarray*, *y: Optional[numpy.ndarray] = None*, *w: Optional[numpy.ndarray] = None*, *ids: Optional[numpy.ndarray] = 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: numpy.ndarray*, *y: Optional[numpy.ndarray] = None*, *w: Optional[numpy.ndarray] = None*, *ids: Optional[numpy.ndarray] = 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: Union[Sequence[int], numpy.ndarray]*, *select_dir: Optional[str] = None*, *select_shard_size: Optional[int] = None*, *output_numpy_dataset: Optional[bool] = False*) → deepchem.data.datasets.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
Get the ids vector for this dataset as a single numpy array.

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

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

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

property memory_cache_size
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() → pandas.core.frame.DataFrame
Return pandas series of label means.

get_label_stds() → pandas.core.frame.DataFrame
Return pandas series of label stds.

static from_dataframe(*df: pandas.core.frame.DataFrame*, *X: Optional[Union[str, Sequence[str]]] = None*, *y: Optional[Union[str, Sequence[str]]] = None*, *w: Optional[Union[str, Sequence[str]]] = None*, *ids: Optional[str] = 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[numpy.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_dataframe() → pandas.core.frame.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: Union[numpy.ndarray, List[str]], y: Optional[Union[numpy.ndarray, List[str]]], w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, ids: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None)

A Dataset that loads data from image files on disk.

__init__(X: Union[numpy.ndarray, List[str]], y: Optional[Union[numpy.ndarray, List[str]]], w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, ids: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = 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() → numpy.ndarray

Get the names of the tasks associated with this dataset.

property X

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

property y

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

property ids

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

property w

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

iterbatches(*batch_size: Optional[int] = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: transformers.Transformer*, ***args*) → deepchem.data.datasets.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: Union[Sequence[int], numpy.ndarray]*, *select_dir: Optional[str] = None*) → deepchem.data.datasets.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: Optional[int] = 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: pandas.core.frame.DataFrame*, *X: Optional[Union[str, Sequence[str]]] = None*, *y: Optional[Union[str, Sequence[str]]] = None*, *w: Optional[Union[str, Sequence[str]]] = None*, *ids: Optional[str] = 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[numpy.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_dataframe() → pandas.core.frame.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

3.6.2 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: deepchem.feat.base_classes.Featurizer*, *feature_field: Optional[str] = None*, *id_field: Optional[str] = None*, *smiles_field: Optional[str] = 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.

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: deepchem.feat.base_classes.Featurizer*, *feature_field: Optional[str] = None*, *id_field: Optional[str] = None*, *smiles_field: Optional[str] = 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: Union[Any, Sequence[Any]]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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: deepchem.feat.base_classes.Featurizer*, *feature_field: Optional[str] = None*, *id_field: Optional[str] = None*, *smiles_field: Optional[str] = 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: deepchem.feat.base_classes.Featurizer*, *feature_field: Optional[str] = None*, *id_field: Optional[str] = None*, *smiles_field: Optional[str] = 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: Union[Any, Sequence[Any]], data_dir: Optional[str] = None, shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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: Optional[List[str]] = None*)

Handles loading of 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.

__init__ (*tasks: Optional[List[str]] = None*)

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.

create_dataset (*inputs: Union[str, Sequence[str], Tuple[Any]], data_dir: Optional[str] = None, shard_size: Optional[int] = 8192, in_memory: bool = False*) → deepchem.data.datasets.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, labels, 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.
- **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: deepchem.feat.base_classes.Featurizer*, *label_field: Optional[str] = None*, *weight_field: Optional[str] = None*, *id_field: Optional[str] = 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: deepchem.feat.base_classes.Featurizer*, *label_field: Optional[str] = None*, *weight_field: Optional[str] = None*, *id_field: Optional[str] = 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: Union[str, Sequence[str]]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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: deepchem.feat.base_classes.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: deepchem.feat.base_classes.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: Union[Any, Sequence[Any]]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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: Optional[deepchem.feat.base_classes.Featurizer] = 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: Optional[deepchem.feat.base_classes.Featurizer] = 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: Union[str, Sequence[str]], data_dir: Optional[str] = None, shard_size: Optional[int] = None*) → deepchem.data.datasets.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*

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: deepchem.feat.base_classes.Featurizer, id_field: Optional[str] = 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
```

__init__(*tasks: List[str], featurizer: deepchem.feat.base_classes.Featurizer, id_field: Optional[str] = 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.

create_dataset (*inputs: Sequence[Any]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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*

3.6.3 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*)

Initialize self. See help(type(self)) for accurate signature.

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].

References

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

Initialize self. See help(type(self)) for accurate signature.

get_pair_edges()

get_pair_features()

get_atom_features()

get_num_atoms()

get_num_features()

__module__ = 'deepchem.feat.mol_graphs'

class GraphData(*node_features: numpy.ndarray*, *edge_index: numpy.ndarray*, *edge_features: Optional[numpy.ndarray] = None*, *node_pos_features: Optional[numpy.ndarray] = None*)

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)

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

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)
>>> graph = GraphData(node_features=node_features, edge_index=edge_index)
```

__init__(*node_features: numpy.ndarray*, *edge_index: numpy.ndarray*, *edge_features: Optional[numpy.ndarray] = None*, *node_pos_features: Optional[numpy.ndarray] = None*)

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].

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.

3.6.4 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

Initialize self. See help(type(self)) for accurate signature.

__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`()` → numpy.ndarray

Get the names of the tasks associated with this dataset.

property X

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

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

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

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: Optional[int] = None*, *epochs: int = 1*, *deterministic: bool = False*, *pad_batches: bool = False*) → Iterator[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: transformers.Transformer*, ***args*) → deepchem.data.datasets.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: Union[Sequence[int], numpy.ndarray]*, *select_dir: Optional[str] = None*) → deepchem.data.datasets.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[numpy.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: Optional[int] = 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() → pandas.core.frame.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: pandas.core.frame.DataFrame*, *X: Optional[Union[str, Sequence[str]]] = None*, *y: Optional[Union[str, Sequence[str]]] = None*, *w: Optional[Union[str, Sequence[str]]] = None*, *ids: Optional[str] = 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().

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: deepchem.feat.base_classes.Featurizer, id_field: Optional[str] = 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: deepchem.feat.base_classes.Featurizer, id_field: Optional[str] = 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: Union[Any, Sequence[Any]]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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: Union[Any, Sequence[Any]]*, *data_dir: Optional[str] = None*, *shard_size: Optional[int] = 8192*) → deepchem.data.datasets.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*

3.7 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.

3.7.1 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.

3.7.2 BACE Dataset

`load_bace_classification`(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGener str]] = ['balancing']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.Transformer]]

Load BACE dataset, classification labels

BACE dataset with classification labels ("class").

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: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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.

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

3.7.3 BBBC Datasets

load_bbbc001(*splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'index', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'index', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.4 BBBP Datasets

BBBP stands for Blood-Brain-Barrier Penetration

load_bbbp(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.5 Cell Counting Datasets

load_cell_counting(*splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = None, transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.6 Chembl Datasets

load_chembl(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], set: str = '5thresh', reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.7 Chembl25 Datasets

load_chembl25(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.8 Clearance Datasets

load_clearance(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['log']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.9 Clintox Datasets

load_clintox(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.10 Delaney Datasets

load_delaney(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.11 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

3.7.12 Freesolv Dataset

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

3.7.13 HIV Datasets

load_hiv(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.14 HOPV Datasets

HOPV stands for the Harvard Organic Photovoltaic Dataset.

load_hopv(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.15 HPPB Datasets

load_hppb(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['log']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.16 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

3.7.17 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

3.7.18 Lipo Datasets

load_lipo(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP'*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.19 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: Union[deepchem.feat.base_classes.Featurizer, str] = ElementPropertyFingerprint[data_source='matminer'], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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: Union[deepchem.feat.base_classes.Featurizer, str] = CGCNNFeaturizer[radius=8.0, max_neighbors=12, step=0.2], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, …], List[transformers.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: Union[deepchem.feat.base_classes.Featurizer, str] = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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: Union[deepchem.feat.base_classes.Featurizer, str] = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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)
```

3.7.20 MUV Datasets

load_muv(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.21 NCI Datasets

load_nci(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.22 PCBA Datasets

load_pcba(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.23 PDBBIND Datasets

load_pdbbind(*featurizer: deepchem.feat.base_classes.ComplexFeaturizer*, *splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random'*, *transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization']*, *reload: bool = True*, *data_dir: Optional[str] = None*, *save_dir: Optional[str] = None*, *pocket: bool = True*, *set_name: str = 'core'*, ***kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.24 PPB Datasets

`load_ppb`(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.25 QM7 Datasets

`load_qm7`(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = CoulombMatrix[max_atoms=23, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.26 QM8 Datasets

load_qm8 (*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = CoulombMatrix[max_atoms=26, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.27 QM9 Datasets

load_qm9 (*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = CoulombMatrix[max_atoms=29, remove_hydrogens=False, randomize=False, upper_tri=False, n_samples=1, seed=None], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.28 SAMPL Datasets

load_sampl(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.29 SIDER Datasets

load_sider(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.30 Thermosol Datasets

load_thermosol(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.31 Tox21 Datasets

load_tox21(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

References

3.7.32 Toxcast Datasets

load_toxcast(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'ECFP', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'scaffold', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['balancing'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, ...], List[transformers.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

3.7.33 USPTO Datasets

load_uspto(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'RxnFeaturizer', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = None, transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, subset: str = 'MIT', sep_reagent: bool = True, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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

3.7.34 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

3.7.35 ZINC15 Datasets

load_zinc15(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = 'OneHot', splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = ['normalization'], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, dataset_size: str = '250K', dataset_dimension: str = '2D', **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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.

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

3.7.36 Platinum Adsorption Dataset

`load_Platinum_Adsorption`(*featurizer: Union[deepchem.feat.base_classes.Featurizer, str] = SineCoulombMatrix[max_atoms=100, flatten=True], splitter: Optional[Union[deepchem.splits.splitters.Splitter, str]] = 'random', transformers: List[Union[deepchem.molnet.load_function.molnet_loader.TransformerGenerator, str]] = [], reload: bool = True, data_dir: Optional[str] = None, save_dir: Optional[str] = None, **kwargs*) → Tuple[List[str], Tuple[deepchem.data.datasets.Dataset, . . .], List[transformers.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
```

3.8 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

3.8.1 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.
- **`ConvMol is an object and not a numpy array`** (`Since`) –
- **`to set dtype to`** (`need`) –
- **`object.`** –

`featurize`(*datapoints: Union[Any, str, Iterable[Any], Iterable[str]]*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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: Optional[int] = 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'>
>>> 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*) → numpy.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: Optional[int] = 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: Optional[List[Any]] = None, atom_labels: Optional[List[int]] = 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: Optional[List[Any]] = None, atom_labels: Optional[List[int]] = 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*) → numpy.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: Union[deepchem.feat.molecule_featurizers.molgan_featurizer.GraphMatrix, Sequence[deepchem.feat.molecule_featurizers.molgan_featurizer.GraphMatrix]], log_every_n: int = 1000*) → numpy.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*) → numpy.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)` → numpy.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'`

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',`

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*) → numpy.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*)

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.

Note: This method requires RDKit to be installed.

Returns **bond_feats** – 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.

Return type np.ndarray

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: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.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__`()

Initialize self. See help(type(self)) for accurate signature.

`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*) → numpy.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*) → numpy.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*) → numpy.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*)

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*)

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).

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → numpy.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: Optional[str] = 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: Optional[str] = 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*) → numpy.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

Returns

Return type np.array

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → numpy.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(*use_fragment=True*, *ipc_avg=True*)

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.

descriptors

List of RDKit descriptor names used in this class.

Type List[str]

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
(208,)
```

__init__(*use_fragment=True*, *ipc_avg=True*)

Initialize this featurizer.

Parameters

- **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.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → numpy.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*) → numpy.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: Optional[int] = 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: Optional[int] = 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*) → numpy.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: numpy.ndarray*) → List[numpy.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*) → numpy.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*) → numpy.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: Optional[int] = 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: Optional[int] = 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*) → numpy.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*) → numpy.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*) → numpy.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: numpy.ndarray*) → List[numpy.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*) → numpy.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*) → numpy.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]*) → numpy.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*) → numpy.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*) → numpy.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: Optional[int] = 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: Optional[int] = 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.

  If max_length is None, no padding is performed and arbitrary length strings are allowed.

`featurize`(*datapoints: Iterable[Any], log_every_n: int = 1000, **kwargs*) → numpy.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: numpy.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

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*) → numpy.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

3.8.2 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).
- **cutoffs and bins** (*Default*) –
- **------------------------** –
- **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: Optional[Iterable[Tuple[str, str]]] = None, log_every_n: int = 100, **kwargs*) → numpy.ndarray

Calculate features for mol/protein complexes.

Parameters **datapoints** (*Iterable[Tuple[str, str]]*) – 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: Optional[Iterable[Tuple[str, str]]] = None*, *log_every_n: int = 100*, ***kwargs*) → numpy.ndarray

Calculate features for mol/protein complexes.

Parameters **datapoints** (*Iterable[Tuple[str, str]]*) – 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

3.8.3 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: Optional[Iterable[str]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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*) → Optional[numpy.ndarray]

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: Optional[Iterable[Union[Dict[str, Any], Any]]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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: Optional[Iterable[Union[Dict[str, Any], Any]]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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],
...              [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',
```

(continues on next page)

(continued from previous page)

```
...                   'A1', 'A1'],
...       "oss": ['-1', '-1', '-1', '-1', '-1', '-1',
...             '-1', '-1', '-1', '-1', '0', '2']
...                   }
... }
>>> featuriser = dc.feat.LCNNFeaturizer(**PRIMITIVE_CELL_INFO)
>>> 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: Optional[Iterable[Union[Dict[str, Any], Any]]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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

3.8.4 MaterialCompositionFeaturizer

3.8.5 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]*)

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]*, *token_ids_1: List[int]*) → List[int]

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]*, *length: int*, *right: bool = True*) → List[int]

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[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(*vocab_path: str*)

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.

3.8.6 Other Featurizers

BertFeaturizer

class `BertFeaturizer`(*tokenizer: transformers.models.bert.tokenization_bert_fast.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: transformers.models.bert.tokenization_bert_fast.BertTokenizerFast*)

Initialize self. See help(type(self)) for accurate signature.

`featurize`(*datapoints: Iterable[Any], log_every_n: int = 1000, **kwargs*) → numpy.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.featurize(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*)

Initialize self. See help(type(self)) for accurate signature.

__len__() → int

Size of the full vocabulary with the added tokens.

add_special_tokens(*special_tokens_dict: Dict[str, Union[str, tokenizers.AddedToken]]*) → 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).

Note,None 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 are carefully handled by the tokenizer (they are never split).
- 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).

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('gpt2') model = GPT2Model.from_pretrained('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: Union[str, tokenizers.AddedToken, List[Union[str, tokenizers.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.

Note,None 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('bert-base-uncased') model = BertModel.from_pretrained('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 additional_special_tokens**

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**

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 the ids of the special tokens(*'<unk>'*, *'<cls>'*, etc.) mapped to class attributes.

**Type** *List[int]*

**property all_special_tokens**

All the special tokens (*'<unk>'*, *'<cls>'*, etc.) mapped to class attributes.

Convert tokens of *tokenizers.AddedToken* type to string.

**Type** *List[str]*

**property all_special_tokens_extended**

All the special tokens (*'<unk>'*, *'<cls>'*, etc.) mapped to class attributes.

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]]*

**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**

The Rust tokenizer used as a backend.

**Type** *tokenizers.implementations.BaseTokenizer*

**batch_decode**(*sequences: Union[List[int], List[List[int]], np.ndarray, torch.Tensor, tf.Tensor]*, *skip_special_tokens: bool = False*, *clean_up_tokenization_spaces: bool = True*, ***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*, defaults to *True*) – Whether or not to clean up the 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: Union[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: Union[bool, str, transformers.file_utils.PaddingStrategy] = False*, *truncation: Union[bool, str, transformers.tokenization_utils_base.TruncationStrategy] = False*, *max_length: Optional[int] = None*, *stride: int = 0*, *is_split_into_words: bool = False*, *pad_to_multiple_of: Optional[int] = None*, *return_tensors: Optional[Union[str, transformers.file_utils.TensorType]] = None*, *return_token_type_ids: Optional[bool] = None*, *return_attention_mask: Optional[bool] = 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*) → transformers.tokenization_utils_base.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 encode the sequences with the special tokens relative to their model.
- **padding** (*bool*, *str* or [*~file_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. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= 7.5 (Volta).

- **return_tensors** (*str* or [*~file_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**

Beginning of sentence token. Log an error if used while not having been set.

**Type** *str*

**property bos_token_id**

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]*

**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**

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**

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: Union[int, List[int]], skip_special_tokens: bool = False*) → Union[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: Union[str, List[str]]*) → Union[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: Optional[List[int]] = 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: Union[int, List[int], np.ndarray, torch.Tensor, tf.Tensor]*, *skip_special_tokens: bool = False*, *clean_up_tokenization_spaces: bool = True*, ***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*, defaults to *True*) – Whether or not to clean up the 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`

The Rust decoder for this tokenizer.

**Type** *tokenizers.decoders.Decoder*

`encode` (*text: Union[str, List[str], List[int]]*, *text_pair: Optional[Union[str, List[str], List[int]]] = None*, *add_special_tokens: bool = True*, *padding: Union[bool, str, transformers.file_utils.PaddingStrategy] = False*, *truncation: Union[bool, str, transformers.tokenization_utils_base.TruncationStrategy] = False*, *max_length: Optional[int] = None*, *stride: int = 0*, *return_tensors: Optional[Union[str, transformers.file_utils.TensorType]] = 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 encode the sequences with the special tokens relative to their model.
- **`padding`** (*bool*, *str* or [*~file_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. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= 7.5 (Volta).

- **return_tensors** (*str* or [*~file_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: Union[str, List[str], List[int]], text_pair: Optional[Union[str, List[str], List[int]]] = None, add_special_tokens: bool = True, padding: Union[bool, str, transformers.file_utils.PaddingStrategy] = False, truncation: Union[bool, str, transformers.tokenization_utils_base.TruncationStrategy] = False, max_length: Optional[int] = None, stride: int = 0, is_split_into_words: bool = False, pad_to_multiple_of: Optional[int] = None, return_tensors: Optional[Union[str, transformers.file_utils.TensorType]] = None, return_token_type_ids: Optional[bool] = None, return_attention_mask: Optional[bool] = 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*) → transformers.tokenization_utils_base.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 encode the sequences with the special tokens relative to their model.
- **padding** (*bool*, *str* or [*~file_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. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= 7.5 (Volta).

- **return_tensors** (*str* or [*~file_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**

End of sentence token. Log an error if used while not having been set.

**Type** *str*

**property eos_token_id**

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*) → numpy.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: Union[str, os.PathLike]*, **init_inputs*, ***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. Valid model ids can be located at the root-level, like *bert-base-uncased*, or namespaced under a user or organization name, like *dbmdz/bert-base-german-cased*.
  - 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.
- **use_auth_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 *transformers-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.
- **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 *use_auth_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('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('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: Optional[List[int]] = 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**

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

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

The maximum combined length of a pair of sentences that can be fed to the model.

Type *int*

property max_len_single_sentence

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: Union[transformers.tokenization_utils_base.BatchEncoding, List[transformers.tokenization_utils_base.BatchEncoding], Dict[str, List[int]], Dict[str, List[List[int]]], List[Dict[str, List[int]]]]*, *padding: Union[bool, str, transformers.file_utils.PaddingStrategy] = True*, *max_length: Optional[int] = None*, *pad_to_multiple_of: Optional[int] = None*, *return_attention_mask: Optional[bool] = None*, *return_tensors: Optional[Union[str, transformers.file_utils.TensorType]] = None*, *verbose: bool = True*) → transformers.tokenization_utils_base.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*)

<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 [*~file_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 [*~file_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

Padding token. Log an error if used while not having been set.

Type *str*

property pad_token_id

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

Id of the padding token type in the vocabulary.

Type *int*

prepare_for_model(*ids: List[int]*, *pair_ids: Optional[List[int]] = None*, *add_special_tokens: bool = True*, *padding: Union[bool, str, transformers.file_utils.PaddingStrategy] = False*, *truncation: Union[bool, str, transformers.tokenization_utils_base.TruncationStrategy] = False*, *max_length: Optional[int] = None*, *stride: int = 0*, *pad_to_multiple_of: Optional[int] = None*, *return_tensors: Optional[Union[str, transformers.file_utils.TensorType]] = None*, *return_token_type_ids: Optional[bool] = None*, *return_attention_mask: Optional[bool] = 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*) → transformers.tokenization_utils_base.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 encode the sequences with the special tokens relative to their model.
- **padding** (*bool*, *str* or [*~file_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. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= 7.5 (Volta).

- **return_tensors** (*str* or [*~file_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: Optional[List[str]] = None*, *max_length: Optional[int] = None*, *max_target_length: Optional[int] = None*, *padding: str = 'longest'*, *return_tensors: Optional[str] = None*, *truncation: bool = True*, ***kwargs*) → transformers.tokenization_utils_base.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 [*~file_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 [*~file_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_path_or_name: Optional[str] = None*, *repo_url: Optional[str] = None*, *use_temp_dir: bool = False*, *commit_message: Optional[str] = None*, *organization: Optional[str] = None*, *private: Optional[bool] = None*, *use_auth_token: Optional[Union[bool, str]] = None*, ***model_card_kwargs*) → str

Upload the tokenizer files to the Model Hub while synchronizing a local clone of the repo in *repo_path_or_name*.

Parameters

- **repo_path_or_name** (*str*, *optional*) – Can either be a repository name for your tokenizer in the Hub or a path to a local folder (in which case the repository will have the name of that local folder). If not specified, will default to the name given by *repo_url* and a local directory with that name will be created.
- **repo_url** (*str*, *optional*) – Specify this in case you want to push to an existing repository in the hub. If unspecified, a new repository will be created in your namespace (unless you specify an *organization*) with *repo_name*.
- **use_temp_dir** (*bool*, *optional*, defaults to *False*) – Whether or not to clone the distant repo in a temporary directory or in *repo_path_or_name* inside the current working directory. This will slow things down if you are making changes in an existing repo since you will need to clone the repo before every push.
- **commit_message** (*str*, *optional*) – Message to commit while pushing. Will default to *"add tokenizer"*.
- **organization** (*str*, *optional*) – Organization in which you want to push your tokenizer (you must be a member of this organization).
- **private** (*bool*, *optional*) – Whether or not the repository created should be private (requires a paying subscription).
- **use_auth_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 *transformers-cli login* (stored in *~/.huggingface*). Will default to *True* if *repo_url* is not specified.

Returns The url of the commit of your tokenizer in the given repository.

Return type *str*

Examples:

```python from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

# Push the tokenizer to your namespace with the name "my-finetuned-bert" and have a local clone in the # *my-finetuned-bert* folder. tokenizer.push_to_hub("my-finetuned-bert")

# Push the tokenizer to your namespace with the name "my-finetuned-bert" with no local clone. tokenizer.push_to_hub("my-finetuned-bert", use_temp_dir=True)

# Push the tokenizer to an organization with the name "my-finetuned-bert" and have a local clone in the # *my-finetuned-bert* folder. tokenizer.push_to_hub("my-finetuned-bert", organization="huggingface")

# Make a change to an existing repo that has been cloned locally in *my-finetuned-bert*. tokenizer.push_to_hub("my-finetuned-bert", repo_url="https://huggingface.co/sgugger/my-finetuned-bert") ```

**sanitize_special_tokens**() → int

Make sure that all the special tokens attributes of the tokenizer (*tokenizer.mask_token*, *tokenizer.cls_token*, etc.) are in the vocabulary.

Add the missing ones to the vocabulary if needed.

**Returns** The number of tokens added in the vocabulary during the operation.

**Return type** *int*

**save_pretrained**(*save_directory: Union[str, os.PathLike]*, *legacy_format: Optional[bool] = None*, *filename_prefix: Optional[str] = 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.

  <Tip warning={true}>

  Using *push_to_hub=True* will synchronize the repository you are pushing to with *save_directory*, which requires *save_directory* to be a local clone of the repo you are pushing to if it's an existing folder. Pass along *temp_dir=True* to use a temporary directory instead.

  </Tip>

Returns The files saved.

Return type A tuple of *str*

`save_vocabulary`(*save_directory: str*, *filename_prefix: Optional[str] = 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`

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

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: transformers.file_utils.PaddingStrategy*, *truncation_strategy: transformers.tokenization_utils_base.TruncationStrategy*, *max_length: int*, *stride: int*, *pad_to_multiple_of: Optional[int]*)

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** ([*~file_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 `transformers.models.roberta.tokenization_roberta.RobertaTokenizer`

property special_tokens_map

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

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: Optional[str] = None*, *add_special_tokens: bool = False*, ***kwargs*) → List[str]

Converts a string in 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*, *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.
- **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** – 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: Optional[List[int]] = None*, *num_tokens_to_remove: int = 0*, *truncation_strategy: Union[str, transformers.tokenization_utils_base.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

Unknown token. Log an error if used while not having been set.

Type *str*

property unk_token_id

Id of the unknown token in the vocabulary. Returns *None* if the token has not been set.

Type *Optional[int]*

property vocab_size

Size of the base vocabulary (without the added tokens).

Type *int*

RxnFeaturizer

class RxnFeaturizer(*tokenizer: transformers.models.roberta.tokenization_roberta_fast.RobertaTokenizerFast*, *sep_reagent: bool*)

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: transformers.models.roberta.tokenization_roberta_fast.RobertaTokenizerFast*, *sep_reagent: bool*)

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.

`featurize`(*datapoints: Iterable[Any]*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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[*deepchem.utils.coordinate_box_utils.CoordinateBox*]*) → numpy.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*) → numpy.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*) → numpy.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

3.8.7 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*) → numpy.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

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.

Note: The subclasses of this class require RDKit to be installed.

featurize(*datapoints*, *log_every_n=1000*, ***kwargs*) → numpy.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: Optional[Iterable[str]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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: Optional[Iterable[Union[Dict[str, Any], Any]]] = None*, *log_every_n: int = 1000*, ***kwargs*) → numpy.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: Optional[Iterable[Tuple[str, str]]] = None*, *log_every_n: int = 100*, ***kwargs*) → numpy.ndarray

Calculate features for mol/protein complexes.

Parameters **datapoints** (*Iterable[Tuple[str, str]]*) – 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

3.9 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

3.9.1 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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

__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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*, ***kwargs*) → List[deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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).

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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: Optional[List[int]] = None*, *test_indices: Optional[List[int]] = 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.

__init__(*valid_indices: Optional[List[int]] = None*, *test_indices: Optional[List[int]] = 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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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*]

3.9.2 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]>
```

(continues on next page)

(continued from previous page)

References

Note: This class requires RDKit to be installed.

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

__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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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.

split(*dataset: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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*]

3.9.3 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: deepchem.data.datasets.Dataset*, *k: int*, *directories: Optional[List[str]] = None*, ***kwargs*) → List[Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *valid_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: int = 1000*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *train_dir: Optional[str] = None*, *test_dir: Optional[str] = None*, *frac_train: float = 0.8*, *seed: Optional[int] = None*, ***kwargs*) → Tuple[deepchem.data.datasets.Dataset, deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *frac_train: float = 0.8*, *frac_valid: float = 0.1*, *frac_test: float = 0.1*, *seed: Optional[int] = None*, *log_every_n: Optional[int] = 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

3.10 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

3.10.1 General Transformers

NormalizationTransformer

class NormalizationTransformer(*transform_X: bool = False*, *transform_y: bool = False*, *transform_w: bool = False*, *dataset: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.ndarray

Not implemented.

transform(*dataset: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[List[int]] = None*, *tasks: Optional[List[str]] = None*, *dataset: Optional[deepchem.data.datasets.Dataset] = 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: Optional[List[int]] = None*, *tasks: Optional[List[str]] = None*, *dataset: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[deepchem.data.datasets.Dataset] = 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_
↪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: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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 *PowerTransformer* doesn't require it to be specified.
- **powers** (list[int], optional (default *[1]*)) – The list of powers of features/labels to compute.

transform_array(*X: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.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: deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.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
>>> 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: deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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: Optional[deepchem.data.datasets.Dataset] = None*, *featurizer: Optional[deepchem.feat.base_classes.Featurizer] = 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: Optional[deepchem.data.datasets.Dataset] = None*, *featurizer: Optional[deepchem.feat.base_classes.Featurizer] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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.

3.10.2 Specified Usecase Transformers

CoulombFitTransformer

class CoulombFitTransformer(*dataset: deepchem.data.datasets.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))
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> print(model.n_features)
12
```

`__init__`(*dataset: deepchem.data.datasets.Dataset*)
Initializes CoulombFitTransformer.

Parameters **`dataset`** (*`dc.data.Dataset`*) – Dataset object to be transformed.

`realize`(*X: numpy.ndarray*) → numpy.ndarray
Randomize features.

Parameters **`X`** (*`np.ndarray`*) – Features

Returns **X** – Randomized features

Return type np.ndarray

`normalize`(*X: numpy.ndarray*) → numpy.ndarray
Normalize features.

Parameters **`X`** (*`np.ndarray`*) – Features

Returns **X** – Normalized features

Return type np.ndarray

`expand`(*X: numpy.ndarray*) → numpy.ndarray
Binarize features.

Parameters **`X`** (*`np.ndarray`*) – Features

Returns **X** – Binarized features

Return type np.ndarray

`X_transform`(*X: numpy.ndarray*) → numpy.ndarray
Perform Coulomb Fit transform on features.

Parameters **`X`** (*`np.ndarray`*) – Features

Returns **X** – Transformed features

Return type np.ndarray

`transform_array`(*X: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.ndarray

Not implemented.

transform(*dataset: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.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: deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.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: numpy.ndarray*) → numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → Union[deepchem.data.datasets.DiskDataset, deepchem.data.datasets.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 *Dataset* object that is transformed.

Return type *DiskDataset* or *NumpyDataset*

untransform(*z: numpy.ndarray*) → numpy.ndarray

Not implemented.

transform_array(*X: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.ndarray

Not implemented.

UG_to_DAG(*sample:* deepchem.feat.mol_graphs.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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.

3.10.3 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: Optional[deepchem.data.datasets.Dataset] = 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: Optional[deepchem.data.datasets.Dataset] = 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: deepchem.data.datasets.Dataset*, *parallel: bool = False*, *out_dir: Optional[str] = None*, ***kwargs*) → deepchem.data.datasets.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*, *y: numpy.ndarray*, *w: numpy.ndarray*, *ids: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.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: numpy.ndarray*) → numpy.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.

3.11 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.

3.11.1 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 is otherwise complete.

As a note about how to read this table, each row describes what's needed to invoke a given model. Some models must be applied with given `Transformer` or `Featurizer` objects. Some models also have custom training methods. You can read off what's needed to train the model from the table below.

Model	Type	Input Type	Transformations	Acceptable Featurizers	Fit Method
AtomicConvModel	Classifier/ Regressor	Tuple		ComplexNeighborListFragmentAtomicCoordinates	fit
ChemCeption	Classifier/ Regressor	Tensor of shape (N, M, c)		SmilesToImage	fit
CNN	Classifier/ Regressor	Tensor of shape (N, c) or (N, M, c) or (N, M, L, c)			fit
DTNNModel	Classifier/ Regressor	Matrix of shape (N, N)		CoulombMatrix	fit
DAGModel	Classifier/ Regressor	ConvMol	DAGTransformer	ConvMolFeaturizer	fit
GraphConvModel	Classifier/ Regressor	ConvMol		ConvMolFeaturizer	fit
MPNNModel	Classifier/ Regressor	WeaveMol		WeaveFeaturizer	fit
MultitaskClassifier	Classifier	Vector of shape (N,)		CircularFingerprint, RDKitDescriptors, CoulombMatrixEig, RdkitGridFeaturizer, BindingPocketFeaturizer, ElementPropertyFingerprint,	fit
MultitaskRegressor	Regressor	Vector of shape (N,)		CircularFingerprint, RDKitDescriptors, CoulombMatrixEig, RdkitGridFeaturizer, BindingPocketFeaturizer, ElementPropertyFingerprint,	fit
MultitaskFitTransformRegressor	Regressor	Vector of shape (N,)	Any	CircularFingerprint, RDKitDescriptors, CoulombMatrixEig, RdkitGridFeaturizer, BindingPocketFeaturizer, ElementPropertyFingerprint,	fit
MultitaskIRVClassifier	Classifier	Vector of shape (N,)	IRVTransformer	CircularFingerprint, RDKitDescriptors, CoulombMatrixEig, RdkitGridFeaturizer, BindingPocketFeaturizer, ElementPropertyFingerprint,	fit

3.11.2 Model

class Model(*model=None*, *model_dir: Optional[str] = None*, ***kwargs*)

Abstract base class for DeepChem models.

__init__(*model=None*, *model_dir: Optional[str] = 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[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: deepchem.data.datasets.Dataset*)

Fits a model on data in a Dataset object.

Parameters **dataset** (Dataset) – the Dataset to train on

predict(*dataset: deepchem.data.datasets.Dataset*, *transformers: List[transformers.Transformer] = []*) → Union[numpy.ndarray, Sequence[numpy.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: deepchem.data.datasets.Dataset*, *metrics: List[deepchem.metrics.metric.Metric]*, *transformers: List[transformers.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.

3.12 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.

3.12.1 SklearnModel

class SklearnModel(*model: sklearn.base.BaseEstimator*, *model_dir: Optional[str] = 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: sklearn.base.BaseEstimator*, *model_dir: Optional[str] = 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: deepchem.data.datasets.Dataset*) → None

Fits scikit-learn model to data.

Parameters **dataset** (`Dataset`) – The *Dataset* to train this model on.

predict_on_batch (*X: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*) → numpy.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: deepchem.data.datasets.Dataset*, *transformers: List[transformers.Transformer] = []*) → Union[numpy.ndarray, Sequence[numpy.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.

3.13 Gradient Boosting Models

Gradient Boosting Models (LightGBM and XGBoost) can be wrapped so they can interact with DeepChem.

3.13.1 GBDTModel

class GBDTModel(*model: sklearn.base.BaseEstimator*, *model_dir: Optional[str] = None*, *early_stopping_rounds: int = 50*, *eval_metric: Optional[Union[Callable, str]] = None*, ***kwargs*)

Wrapper class that wraps GBDT models as DeepChem models.

This class supports LightGBM/XGBoost models.

__init__(*model: sklearn.base.BaseEstimator*, *model_dir: Optional[str] = None*, *early_stopping_rounds: int = 50*, *eval_metric: Optional[Union[Callable, str]] = 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, Callbale]*) – 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: deepchem.data.datasets.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: deepchem.data.datasets.Dataset*, *valid_dataset: deepchem.data.datasets.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.

3.14 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.

3.14.1 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

3.14.2 Optimizers

class Optimizer(*learning_rate: Union[float,* deepchem.models.optimizers.LearningRateSchedule*]*)

An algorithm for optimizing a model.

This is an abstract class. Subclasses represent specific optimization algorithms.

__init__(*learning_rate: Union[float,* deepchem.models.optimizers.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: Union[float,* deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-08*)

The Adam optimization algorithm.

`__init__`(*learning_rate: Union[float, deepchem.models.optimizers.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 Adam algorithm
- **`beta2`** (*`float`*) – a parameter of the Adam algorithm
- **`epsilon`** (*`float`*) – a parameter of the Adam algorithm

`class AdamW`(*learning_rate: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001, weight_decay: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001, weight_decay: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001*, *momentum: float = 0.0*, *decay: float = 0.9*, *epsilon: float = 1e-10*)

RMSProp Optimization algorithm.

__init__(*learning_rate: Union[float, deepchem.models.optimizers.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: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001*)

The gradient descent optimization algorithm.

__init__(*learning_rate: Union[float, deepchem.models.optimizers.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*) –
- **learning rate** (*initial*) –
- **decay_steps** (*int*) –
- **of steps to decay over** (*number*) –
- **num_periods** (*number of periods in the cosine part of the decay*) –

3.15 Keras Models

DeepChem extensively uses Keras to build deep learning models.

3.15.1 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(...)

# 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(...)
```

(continues on next page)

(continued from previous page)

```
model = KerasModel(..., wandb_logger=logger)
model.fit(..., callbacks=[vc])
logger.finish()
```

class KerasModel(*model: keras.engine.training.Model*, *loss: Union[deepchem.models.losses.Loss, Callable[[List, List, List], Any]]*, *output_types: Optional[List[str]] = None*, *batch_size: int = 100*, *model_dir: Optional[str] = None*, *learning_rate: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001*, *optimizer: Optional[deepchem.models.optimizers.Optimizer] = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *wandb_logger: Optional[deepchem.models.wandblogger.WandbLogger] = 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: keras.engine.training.Model, loss: Union[deepchem.models.losses.Loss, Callable[[List, List, List], Any]], output_types: Optional[List[str]] = None, batch_size: int = 100, model_dir: Optional[str] = None, learning_rate: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001, optimizer: Optional[deepchem.models.optimizers.Optimizer] = None, tensorboard: bool = False, wandb: bool = False, log_frequency: int = 100, wandb_logger: Optional[deepchem.models.wandblogger.WandbLogger] = 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: deepchem.data.datasets.Dataset, nb_epoch: int = 10, max_checkpoints_to_keep: int = 5, checkpoint_interval: int = 1000, deterministic: bool = False, restore: bool = False, variables: Optional[List[tensorflow.python.ops.variables.Variable]] = None, loss: Optional[Callable[[List, List, List], Any]] = None, callbacks: Union[Callable, List[Callable]] = [], all_losses: Optional[List[float]] = 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

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: Optional[List[tensorflow.python.ops.variables.Variable]] = None, loss: Optional[Callable[[List, List, List], Any]] = None, callbacks: Union[Callable, List[Callable]] = [], all_losses: Optional[List[float]] = 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) 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

Return type The average loss over the most recent checkpoint interval

fit_on_batch(*X: Sequence*, *y: Sequence*, *w: Sequence*, *variables: Optional[List[tensorflow.python.ops.variables.Variable]] = None*, *loss: Optional[Callable[[List, List, List], Any]] = None*, *callbacks: Union[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

Return type the loss on the batch

predict_on_generator(*generator: Iterable[Tuple[Any, Any, Any]]*, *transformers: List[transformers.Transformer] = []*, *outputs: Optional[Union[tensorflow.python.framework.ops.Tensor, Sequence[tensorflow.python.framework.ops.Tensor]]] = None*, *output_types: Optional[Union[str, Sequence[str]]] = None*) → Union[numpy.ndarray, Sequence[numpy.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

predict_on_batch(*X: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*, *transformers: List[transformers.Transformer] = []*, *outputs: Optional[Union[tensorflow.python.framework.ops.Tensor, Sequence[tensorflow.python.framework.ops.Tensor]]] = None*) → Union[numpy.ndarray, Sequence[numpy.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*

predict_uncertainty_on_batch(*X: Sequence*, *masks: int = 50*) → Union[Tuple[numpy.ndarray, numpy.ndarray], Sequence[Tuple[numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *transformers: List[transformers.Transformer] = []*, *outputs: Optional[Union[tensorflow.python.framework.ops.Tensor, Sequence[tensorflow.python.framework.ops.Tensor]]] = None*, *output_types: Optional[List[str]] = None*) → Union[numpy.ndarray, Sequence[numpy.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: deepchem.data.datasets.Dataset*) → Union[numpy.ndarray, Sequence[numpy.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: deepchem.data.datasets.Dataset*, *masks: int = 50*) → Union[Tuple[numpy.ndarray, numpy.ndarray], Sequence[Tuple[numpy.ndarray, numpy.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[deepchem.metrics.metric.Metric], transformers: List[transformers.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: numpy.ndarray*) → Union[numpy.ndarray, Sequence[numpy.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

Returns

Return type the Jacobian matrix, or a list of matrices

default_generator(*dataset: deepchem.data.datasets.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: Optional[str] = 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: Optional[str] = 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: Optional[str] = None*, *model_dir: Optional[str] = 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: deepchem.models.keras_model.KerasModel*, *assignment_map: Optional[Dict[Any, Any]] = None*, *value_map: Optional[Dict[Any, Any]] = None*, *checkpoint: Optional[str] = None*, *model_dir: Optional[str] = None*, *include_top: bool = True*, *inputs: Optional[Sequence[Any]] = 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 pretrained 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

3.15.2 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)

3.15.3 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])*

3.15.4 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: deepchem.data.datasets.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])*

3.15.5 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.

3.15.6 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

Return type The average loss over the most recent checkpoint interval

fit_task(*dataset*, *task*, *nb_epoch=10*, *max_checkpoints_to_keep=5*, *checkpoint_interval=1000*, *deterministic=False*, *restore=False*, ***kwargs*)
Fit one task.

3.15.7 WeaveModel

class WeaveModel(*n_tasks: int, n_atom_feat: Union[int, Sequence[int]] = 75, n_pair_feat: Union[int, Sequence[int]] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: List[int] = [2000, 100], conv_weight_init_stddevs: Union[float, Sequence[float]] = 0.03, weight_init_stddevs: Union[float, Sequence[float]] = 0.01, bias_init_consts: Union[float, Sequence[float]] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.25, final_conv_activation_fn: Optional[Union[Callable, str]] = <function tanh>, activation_fns: Union[Callable, str, Sequence[Union[Callable, str]]] = <function relu>, batch_normalize: bool = True, batch_normalize_kwargs: 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: Union[int, Sequence[int]] = 75, n_pair_feat: Union[int, Sequence[int]] = 14, n_hidden: int = 50, n_graph_feat: int = 128, n_weave: int = 2, fully_connected_layer_sizes: List[int] = [2000, 100], conv_weight_init_stddevs: Union[float, Sequence[float]] = 0.03, weight_init_stddevs: Union[float, Sequence[float]] = 0.01, bias_init_consts: Union[float, Sequence[float]] = 0.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.25, final_conv_activation_fn: Optional[Union[Callable, str]] = <function tanh>, activation_fns: Union[Callable, str, Sequence[Union[Callable, str]]] = <function relu>, batch_normalize: bool = True, batch_normalize_kwargs: 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: deepchem.data.datasets.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*.

Returns

Return type Iterator which walks over the batches

3.15.8 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])*

3.15.9 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

3.15.10 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])*

3.15.11 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 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 **[1]_**

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])*

3.15.12 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]_. 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))
```

(continues on next page)

(continued from previous page)

```
>> # 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)
>> 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() → keras.engine.training.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). In case of training, not flag is necessary.

create_discriminator() → keras.engine.training.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: Optional[List] = None*, *conditional_inputs: List = []*, *generator_index: int = 0*) → List[deepchem.feat.molecule_featurizers.molgan_featurizer.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]

3.15.13 ScScoreModel

class ScScoreModel(*n_features*, *layer_sizes=[300, 300, 300]*, *dropouts=0.0*, ***kwargs*)

https://pubs.acs.org/doi/abs/10.1021/acs.jcim.7b00622 Several definitions of molecular complexity exist to facilitate prioritization of lead compounds, to identify diversity-inducing and complexifying reactions, and to guide retrosynthetic searches. In this work, we focus on synthetic complexity and reformalize its definition to correlate with the expected number of reaction steps required to produce a target molecule, with implicit knowledge about what compounds are reasonable starting materials. We train a neural network model on 12 million reactions from the Reaxys database to impose a pairwise inequality constraint enforcing the premise of this definition: 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.

Our model here actually uses hingeloss instead of the shifted relu loss in https://github.com/connorcoley/scscore.

This could cause issues differentiation issues with compounds that are "close" to each other in "complexity"

__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])*

3.15.14 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

3.15.15 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.

Returns

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.

Returns

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.

Returns

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.

Returns

Return type A Tensor equal to the loss function to use for optimizing the discriminator.

3.15.16 CNN

class CNN (*n_tasks, n_features, dims, layer_filters=[100], kernel_size=5, strides=1, 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>, dense_layer_size=1000, pool_type='max', mode='classification', n_classes=2, uncertainty=False, residual=False, padding='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 dense 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.

__init__ (*n_tasks, n_features, dims, layer_filters=[100], kernel_size=5, strides=1, 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>, dense_layer_size=1000, pool_type='max', mode='classification', n_classes=2, uncertainty=False, residual=False, padding='valid', **kwargs*)
Create a CNN.

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
- **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 loat*) – 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 probablity 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** (*list or object*) – the Tensorflow 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.
- **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*) – the type of 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])*

3.15.17 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

[2] Kim, Yoon. "Convolutional neural networks for sentence classification." arXiv preprint arXiv:1408.5882 (2014).

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

3.15.18 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: 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: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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: 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: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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.

3.15.19 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])*

3.15.20 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])*

3.15.21 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: deepchem.models.normalizing_flows.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: deepchem.models.normalizing_flows.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: Union[tensorflow.python.framework.ops.Tensor, Sequence[tensorflow.python.framework.ops.Tensor]]*) → tensorflow.python.framework.ops.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.

Returns

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.

3.16 PyTorch Models

DeepChem supports the use of PyTorch to build deep learning models.

3.16.1 TorchModel

You can wrap an arbitrary `torch.nn.Module` in a `TorchModel` object.

class TorchModel(*model: torch.nn.modules.module.Module*, *loss: Union[deepchem.models.losses.Loss, Callable[[List, List, List], Any]]*, *output_types: Optional[List[str]] = None*, *batch_size: int = 100*, *model_dir: Optional[str] = None*, *learning_rate: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001*, *optimizer: Optional[deepchem.models.optimizers.Optimizer] = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *device: Optional[torch.device] = None*, *regularization_loss: Optional[Callable] = None*, *wandb_logger: Optional[deepchem.models.wandblogger.WandbLogger] = 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: torch.nn.modules.module.Module*, *loss: Union[deepchem.models.losses.Loss, Callable[[List, List, List], Any]]*, *output_types: Optional[List[str]] = None*, *batch_size: int = 100*, *model_dir: Optional[str] = None*, *learning_rate: Union[float, deepchem.models.optimizers.LearningRateSchedule] = 0.001*, *optimizer: Optional[deepchem.models.optimizers.Optimizer] = None*, *tensorboard: bool = False*, *wandb: bool = False*, *log_frequency: int = 100*, *device: Optional[torch.device] = None*, *regularization_loss: Optional[Callable] = None*, *wandb_logger: Optional[deepchem.models.wandblogger.WandbLogger] = 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: deepchem.data.datasets.Dataset*, *nb_epoch: int = 10*, *max_checkpoints_to_keep: int = 5*, *checkpoint_interval: int = 1000*, *deterministic: bool = False*, *restore: bool = False*, *variables: Optional[List[torch.nn.parameter.Parameter]] = None*, *loss: Optional[Callable[[List, List, List], Any]] = None*, *callbacks: Union[Callable, List[Callable]] = []*, *all_losses: Optional[List[float]] = 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) 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

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: Optional[List[torch.nn.parameter.Parameter]] = None, loss: Optional[Callable[[List, List, List], Any]] = None, callbacks: Union[Callable, List[Callable]] = [], all_losses: Optional[List[float]] = 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) 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

Return type The average loss over the most recent checkpoint interval

fit_on_batch(*X: Sequence, y: Sequence, w: Sequence, variables: Optional[List[torch.nn.parameter.Parameter]] = None, loss: Optional[Callable[[List, List, List], Any]] = None, callbacks: Union[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.

Returns

Return type the loss on the batch

predict_on_generator(*generator: Iterable[Tuple[Any, Any, Any]], transformers: List[transformers.Transformer] = [], output_types: Optional[Union[str, Sequence[str]]] = None*) → Union[numpy.ndarray, Sequence[numpy.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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], transformers: List[transformers.Transformer] = []*) → Union[numpy.ndarray, Sequence[numpy.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*) → Union[Tuple[numpy.ndarray, numpy.ndarray], Sequence[Tuple[numpy.ndarray, numpy.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: deepchem.data.datasets.Dataset*, *transformers: List[transformers.Transformer] = []*, *output_types: Optional[List[str]] = None*) → Union[numpy.ndarray, Sequence[numpy.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: deepchem.data.datasets.Dataset*) → Union[numpy.ndarray, Sequence[numpy.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: deepchem.data.datasets.Dataset*, *masks: int = 50*) → Union[Tuple[numpy.ndarray, numpy.ndarray], Sequence[Tuple[numpy.ndarray, numpy.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[deepchem.metrics.metric.Metric], transformers: List[transformers.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: numpy.ndarray*) → Union[numpy.ndarray, Sequence[numpy.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

Returns

Return type the Jacobian matrix, or a list of matrices

default_generator(*dataset: deepchem.data.datasets.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: Optional[str] = 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: Optional[str] = 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: Optional[str] = None*, *model_dir: Optional[str] = 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.

get_global_step() → int

Get the number of steps of fitting that have been performed.

load_from_pretrained(*source_model: deepchem.models.torch_models.torch_model.TorchModel*, *assignment_map: Optional[Dict[Any, Any]] = None*, *value_map: Optional[Dict[Any, Any]] = None*, *checkpoint: Optional[str] = None*, *model_dir: Optional[str] = None*, *include_top: bool = True*, *inputs: Optional[Sequence[Any]] = 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 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
- **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 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.

3.16.2 MultitaskRegressor

class MultitaskRegressor(*n_tasks: int, n_features: int, layer_sizes: Sequence[int] = [1000], weight_init_stddevs: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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: deepchem.data.datasets.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])*

3.16.3 MultitaskFitTransformRegressor

class MultitaskFitTransformRegressor(*n_tasks: int*, *n_features: int*, *fit_transformers: Sequence[transformers.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[transformers.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: deepchem.data.datasets.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[transformers.Transformer] = [], output_types: Optional[Union[str, Sequence[str]]] = None*) → Union[numpy.ndarray, Sequence[numpy.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

3.16.4 MultitaskClassifier

class MultitaskClassifier(*n_tasks: int, n_features: int, layer_sizes: Sequence[int] = [1000], weight_init_stddevs: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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: Union[float, Sequence[float]] = 0.02, bias_init_consts: Union[float, Sequence[float]] = 1.0, weight_decay_penalty: float = 0.0, weight_decay_penalty_type: str = 'l2', dropouts: Union[float, Sequence[float]] = 0.5, activation_fns: Union[Callable, str, Sequence[Union[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: deepchem.data.datasets.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])*

3.16.5 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.

3.16.6 GATModel

class GATModel(*n_tasks: int, graph_attention_layers: Optional[list] = None, n_attention_heads: int = 8, agg_modes: Optional[list] = 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: Optional[list] = None, n_attention_heads: int = 8, agg_modes: Optional[list] = 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.

3.16.7 GCNModel

class GCNModel(*n_tasks: int*, *graph_conv_layers: Optional[list] = 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: Optional[list] = 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.

3.16.8 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.

3.16.9 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.

3.16.10 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.

__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.

3.16.11 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],
..               [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.

3.17 Jax Models

DeepChem supports the use of Jax to build deep learning models.

3.17.1 JaxModel

3.17.2 PinnModel

3.18 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.

3.18.1 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*)

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.

Returns

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: Optional[Callable] = 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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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: 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: 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: Tuple = (128, 64)*, *activation: 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: List = [(128, 64), 128], activation: 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).

[2] Vinyals, Oriol, Samy Bengio, and Manjunath Kudlur. "Order matters: Sequence to sequence for sets." arXiv preprint arXiv:1511.06391 (2015).

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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.

Returns

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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)

  A are the adjacency matrixes for each graph Shape (batch, from_vertex, adj_matrix, to_vertex)

  mask is optional, to be used when not every graph has the same number of vertices

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)

  A are the adjacency matrixes for each graph Shape (batch, from_vertex, adj_matrix, to_vertex)

  mask is optional, to be used when not every graph has the same number of vertices
- **Returns** (*tf.tensor*) –
- **a tf.tensor with a graph convolution applied** (*Returns*) –
- **shape will be (batch** (*The*) –
- **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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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],
...                          [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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

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 (optional, 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.

This is typically used to create the weights of *Layer* subclasses.

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

3.18.2 Torch Layers

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: torch.Tensor*) → torch.Tensor

Defines 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
>>> 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: torch.Tensor*, *mask: torch.Tensor*, *adj_matrix: torch.Tensor*, *distance_matrix: torch.Tensor*, *sa_dropout_p: float = 0.0*) → torch.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: torch.Tensor*, *key: torch.Tensor*, *value: torch.Tensor*, *mask: torch.Tensor*, *adj_matrix: torch.Tensor*, *distance_matrix: torch.Tensor*, *dropout_p: float = 0.0*, *eps: float = 1e-06*, *inf: float = 1000000000000.0*) → torch.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 which establishes a residual connection, as used in the Molecular Attention Transformer [1]_.

The SublayerConnection layer is a residual layer which is then passed through Layer Normalization. The residual connection is established by computing the dropout-adjusted layer output of a normalized tensor and adding this to the original input tensor.

References

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: torch.Tensor*, *output: torch.Tensor*) → torch.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*)

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.

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*)

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.

`forward`(*x: torch.Tensor*) → torch.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: torch.Tensor*) → torch.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: torch.Tensor*, *mask: torch.Tensor*) → torch.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 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)
>>> tf.reduce_sum(diff) == 0 # True
<tf.Tensor: shape=(), dtype=bool, numpy=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)*.

```
>>> tf.reduce_sum(cos_sim_orth) == 0 # True
<tf.Tensor: shape=(), dtype=bool, numpy=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

3.18.3 Jax Layers

3.19 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.

3.19.1 Metric Utilities

Metric utility functions allow for some common manipulations such as switching to/from one-hot representations.

`to_one_hot` (*y: numpy.ndarray*, *n_classes: int = 2*) → numpy.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: numpy.ndarray*, *axis: int = 1*) → numpy.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

3.19.2 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: Optional[numpy.ndarray]*, *n_samples: int*, *n_tasks: int*) → numpy.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: numpy.ndarray*, *mode: Optional[str] = None*, *n_tasks: Optional[int] = None*, *n_classes: Optional[int] = None*) → numpy.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: numpy.ndarray, mode: Optional[str] = None, n_tasks: Optional[int] = None, n_classes: Optional[int] = 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: numpy.ndarray, classification_handling_mode: Optional[str], threshold_value: Optional[float] = None*) → numpy.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

3.19.3 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, shape = [n_samples]*) – Ground truth (correct) target values.
- **y_pred** (*array, 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.

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 to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. 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** (*str or int, default=1*) – The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that 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 or 1, default="warn"*) – Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

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

```
>>> 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. ])
>>> # 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'*)

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). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

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'.

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)
```

(continues on next page)

(continued from previous page)

```
0.0
>>> y_true = [1, 2, 3]
>>> y_pred = [3, 2, 1]
>>> r2_score(y_true, y_pred)
-3.0
```

mean_squared_error(*y_true*, *y_pred*, ***, *sample_weight=None*, *multioutput='uniform_average'*, *squared=True*)

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.

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 = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred, squared=False)
0.612...
>>> 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, squared=False)
0.822...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
```

(continues on next page)

(continued from previous page)

```
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.

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 to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. 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** (*str or int, default=1*) – The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that 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 or 1, default="warn"*) – Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

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

```
>>> 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.        ])
>>> # multilabel classification
>>> y_true = [[0, 0, 0], [1, 1, 1], [0, 1, 1]]
>>> 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*)

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.

Read more in the User Guide.

Parameters

- **y_true** (*ndarray 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** (*ndarray 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 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.

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. 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([0.66666667, 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 0.5, 0.5, 0. ])
>>> thresholds
array([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** (*ndarray of shape (n,)*) – x coordinates. These must be either monotonic increasing or monotonic decreasing.
- **y** (*ndarray of shape, (n,)*) – y coordinates.

Returns auc

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`.

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 to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

- **pos_label** (*str or int, default=1*) – The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that 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

Return type float (if average is not None) or array of floats, shape = [n_unique_labels]

See also:

accuracy_score, *f1_score*, `multilabel_confusion_matrix`

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 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:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the `average` parameter.

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 to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. 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** (*str or int, default=1*) – The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that 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 or 1, default="warn"*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to "warn", this acts as 0, but warnings are also raised.

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`, `precision_recall_fscore_support`, *jaccard_score*, `multilabel_confusion_matrix`

References

Examples

```
>>> 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. ])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> f1_score(y_true, y_pred, zero_division=1)
1.0...
>>> # 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])
```

Notes

When `true positive + false positive == 0`, precision is undefined. When `true positive + false negative == 0`, recall is undefined. In such cases, by default the metric will be set to 0, as will f-score, and `UndefinedMetricWarning` will be raised. This behavior can be modified with `zero_division`.

`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.

  `'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

Return type float

References

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.

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...
```

[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.

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
```

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** (*1d array-like*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like*) – 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
```

pearson_r2_score(*y: numpy.ndarray*, *y_pred: numpy.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: numpy.ndarray*, *y_pred: numpy.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: numpy.ndarray*, *y_pred: numpy.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: numpy.ndarray*, *y_pred: numpy.ndarray*) → 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: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Computes RMS error.

mae_score(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Computes MAE.

kappa_score(*y1*, *y2*, ***, *labels=None*, *weights=None*, *sample_weight=None*)

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 of shape (n_samples,)*) – Labels assigned by the first annotator.
- **y2** (*array 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

bedroc_score(*y_true: numpy.ndarray*, *y_pred: numpy.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: numpy.ndarray*, *y_pred: numpy.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: numpy.ndarray*, *motif_names: List[str]*, *max_scores: Optional[int] = None*, *return_positions: bool = False*, *GC_fraction: float = 0.4*) → numpy.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: numpy.ndarray*, *pssm: numpy.ndarray*) → numpy.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: deepchem.models.models.Model*, *encoded_sequences: numpy.ndarray*) → numpy.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

3.19.4 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: Optional[Callable[[. . .], Any]] = None*, *name: Optional[str] = None*, *threshold: Optional[float] = None*, *mode: Optional[str] = None*, *n_tasks: Optional[int] = None*, *classification_handling_mode: Optional[str] = None*, *threshold_value: Optional[float] = 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: Optional[Callable[[…], Any]] = None*, *name: Optional[str] = None*, *threshold: Optional[float] = None*, *mode: Optional[str] = None*, *n_tasks: Optional[int] = None*, *classification_handling_mode: Optional[str] = None*, *threshold_value: Optional[float] = 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*, *y_pred: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*, *w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]], Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None*, *n_tasks: Optional[int] = 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: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]] Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[nump bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], y_pred: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]] Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[nump bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]], w: Optional[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], numpy.typing._array_like._SupportsArray[numpy.dtype], Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]], Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[numpy.dtype]]]] Sequence[Sequence[Sequence[Sequence[numpy.typing._array_like._SupportsArray[nump bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]] = None, n_samples: Optional[int] = 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

3.20 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.

3.20.1 Hyperparameter Optimization API

class HyperparamOpt(*model_builder: Callable[[. . .], deepchem.models.models.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[[. . .], deepchem.models.models.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: deepchem.data.datasets.Dataset, valid_dataset: deepchem.data.datasets.Dataset, metric: deepchem.metrics.metric.Metric, output_transformers: List[transformers.Transformer] = [], nb_epoch: int = 10, use_max: bool = True, logdir: Optional[str] = None, **kwargs*) → Tuple[deepchem.models.models.Model, Dict, Dict]

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.

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*]

3.20.2 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[[. . .], deepchem.models.models.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']
...   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: deepchem.data.datasets.Dataset*, *valid_dataset: deepchem.data.datasets.Dataset*, *metric: deepchem.metrics.metric.Metric*, *output_transformers: List[transformers.Transformer] = []*, *nb_epoch: int = 10*, *use_max: bool = True*, *logdir: Optional[str] = None*, *logfile: Optional[str] = None*, ***kwargs*)

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 None)*) – 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 *(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*]

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.

3.20.3 Gaussian Process Hyperparameter Optimization

class GaussianProcessHyperparamOpt(*model_builder: Callable[[. . .], deepchem.models.models.Model]*)

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'>

Notes

This class requires pyGPGO to be installed.

hyperparam_search(*params_dict: Dict*, *train_dataset: deepchem.data.datasets.Dataset*, *valid_dataset: deepchem.data.datasets.Dataset*, *metric: deepchem.metrics.metric.Metric*, *output_transformers: List[transformers.Transformer] = []*, *nb_epoch: int = 10*, *use_max: bool = True*, *logdir: Optional[str] = None*, *max_iter: int = 20*, *search_range: Union[int, float, Dict] = 4*, *logfile: Optional[str] = None*, ***kwargs*)

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.
- **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]]
- **logfile** (*str, optional (default None)*) – 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/.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*]

3.21 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.

3.21.1 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 Tensorflow 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.

3.21.2 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*

3.22 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.

3.22.1 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)*

3.22.2 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.

3.22.3 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.

Returns

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.

Returns

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*)

Initialize self. See help(type(self)) for accurate signature.

3.22.4 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.

Returns

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.

Returns

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*)

Initialize self. See help(type(self)) for accurate signature.

3.23 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.

3.23.1 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: Optional[deepchem.models.models.Model] = 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: Optional[deepchem.models.models.Model] = 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[*deepchem.utils.coordinate_box_utils.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[*deepchem.utils.coordinate_box_utils.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*]

3.23.2 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: Optional[numpy.ndarray] = None*, *box_dims: Optional[numpy.ndarray] = None*, *exhaustiveness: int = 10*, *num_modes: int = 9*, *num_pockets: Optional[int] = None*, *out_dir: Optional[str] = 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.

Returns

Return type A list of molecular complexes in energetically favorable poses.

class VinaPoseGenerator(*pocket_finder: Optional[*deepchem.dock.binding_pocket.BindingPocketFinder*] = 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.

__init__(*pocket_finder: Optional[*deepchem.dock.binding_pocket.BindingPocketFinder*] = 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: Optional[numpy.ndarray] = None, box_dims: Optional[numpy.ndarray] = None, exhaustiveness: int = 10, num_modes: int = 9, num_pockets: Optional[int] = None, out_dir: Optional[str] = None, generate_scores: Optional[bool] = False, **kwargs*) → Union[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)* 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 a list of binding free energies predicted by Vina.

Return type Tuple[*docked_poses*, *scores*] or *docked_poses*

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: Optional[numpy.ndarray] = None*, *box_dims: Optional[numpy.ndarray] = None*, *exhaustiveness: int = 10*, *num_modes: int = 9*, *num_pockets: Optional[int] = None*, *out_dir: Optional[str] = None*, *generate_scores: bool = True*, ***kwargs*) → Union[Tuple[List[Tuple[Any, Any]], numpy.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*

3.23.3 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:* deepchem.dock.pose_generation.PoseGenerator, *featurizer: Optional[deepchem.feat.base_classes.ComplexFeaturizer] = None*, *scoring_model: Optional[deepchem.models.models.Model] = 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:* deepchem.dock.pose_generation.PoseGenerator, *featurizer: Optional[deepchem.feat.base_classes.ComplexFeaturizer] = None*, *scoring_model: Optional[deepchem.models.models.Model] = 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: Optional[numpy.ndarray] = None*, *box_dims: Optional[numpy.ndarray] = None*, *exhaustiveness: int = 10*, *num_modes: int = 9*, *num_pockets: Optional[int] = None*, *out_dir: Optional[str] = None*, *use_pose_generator_scores: bool = False*) → Union[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*]

3.23.4 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: numpy.ndarray*, *coords2: numpy.ndarray*) → numpy.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: numpy.ndarray*, *x: numpy.ndarray*, *cutoff=8.0*) → numpy.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: numpy.ndarray*, *w: float*, *Nrot: int*) → numpy.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: numpy.ndarray*) → numpy.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: numpy.ndarray*) → numpy.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: numpy.ndarray*) → numpy.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: numpy.ndarray*) → numpy.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: numpy.ndarray*) → numpy.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: numpy.ndarray*, *coords2: numpy.ndarray*, *weights: numpy.ndarray*, *wrot: float*, *Nrot: int*) → numpy.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

3.24 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.

3.24.1 Data Utilities

Array Utilities

pad_array(*x: numpy.ndarray*, *shape: Union[Tuple, int]*, *fill: float = 0.0*, *both: bool = False*) → numpy.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: Optional[str] = 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: Optional[str] = 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: Optional[str] = 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: Optional[int] = 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: Optional[int] = None*) → Iterator[pandas.core.frame.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: Optional[int] = None*) → Iterator[pandas.core.frame.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: Optional[int] = None*) → Iterator[pandas.core.frame.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, Optional[Tuple[deepchem.data.datasets.DiskDataset, deepchem.data.datasets.DiskDataset, deepchem.data.datasets.DiskDataset]], List[transformers.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: deepchem.data.datasets.DiskDataset*, *valid: deepchem.data.datasets.DiskDataset*, *test: deepchem.data.datasets.DiskDataset*, *transformers: List[transformers.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`

3.24.2 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'.

[1] http://rdkit.org/docs/GettingStartedInPython.html#working-with-3d-molecules

[2] http://pubs.acs.org/doi/full/10.1021/ci2004658

- **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: Optional[int] = 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*) → numpy.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*) → numpy.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*)

Initialize self. See help(type(self)) for accurate signature.

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

Returns

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.

Returns

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*) –

Returns

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.:

3.24.3 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: numpy.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: numpy.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[*deepchem.utils.fragment_utils.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`() → numpy.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: numpy.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: numpy.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() → numpy.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: numpy.ndarray*, *mol: Union[Any,* deepchem.utils.fragment_utils.MolecularFragment*]*) → Tuple[numpy.ndarray, *deepchem.utils.fragment_utils.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[*deepchem.utils.fragment_utils.MolecularFragment*]*) → Optional[*deepchem.utils.fragment_utils.MolecularFragment*]

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[numpy.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[numpy.ndarray, Any]]*, *cutoff: float = 4.5*) → List[Tuple[numpy.ndarray, *deepchem.utils.fragment_utils.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*]]

3.24.4 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: deepchem.utils.coordinate_box_utils.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: deepchem.utils.coordinate_box_utils.CoordinateBox*, *box2: deepchem.utils.coordinate_box_utils.CoordinateBox*) → *deepchem.utils.coordinate_box_utils.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[*deepchem.utils.coordinate_box_utils.CoordinateBox*]*, *threshold: float = 0.8*) → List[*deepchem.utils.coordinate_box_utils.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: numpy.ndarray*, *pad: float = 5.0*) → List[*deepchem.utils.coordinate_box_utils.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)
```

3.24.5 Evaluation Utils

class Evaluator(*model*, *dataset: deepchem.data.datasets.Dataset*, *transformers: List[transformers.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: deepchem.data.datasets.Dataset*, *transformers: List[transformers.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: numpy.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: Union[deepchem.metrics.metric.Metric, Callable[[...], Any], List[deepchem.metrics.metric.Metric], List[Callable[[...], Any]]]*, *csv_out: Optional[str] = None*, *stats_out: Optional[str] = None*, *per_task_metrics: bool = False*, *use_sample_weights: bool = False*, *n_classes: int = 2*) → Union[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[transformers.Transformer]*, *labels: Optional[List] = None*, *weights: Optional[List] = 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[transformers.Transformer]*, *labels: Optional[List] = None*, *weights: Optional[List] = 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: Union[deepchem.metrics.metric.Metric, Callable[[...], Any], List[deepchem.metrics.metric.Metric], List[Callable[[...], Any]]]*, *per_task_metrics: bool = False*, *use_sample_weights: bool = False*, *n_classes: int = 2*) → Union[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: numpy.ndarray*, *y: numpy.ndarray*) → numpy.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

3.24.6 Genomic Utilities

seq_one_hot_encode (*sequences*, *letters: str = 'ATCGN'*) → numpy.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'*) → numpy.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.

3.24.7 Geometry Utilities

unit_vector(*vector: numpy.ndarray*) → numpy.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: numpy.ndarray*, *vector_j: numpy.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() → numpy.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``()` → numpy.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: numpy.ndarray*, *vector_j: numpy.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

3.24.8 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: Optional[Dict[int, str]] = None*, *size: int = 1024*, *feature_list: Optional[List] = None*) → numpy.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

3.24.9 Voxel Utils

`convert_atom_to_voxel`(*coordinates: numpy.ndarray*, *atom_index: int*, *box_width: float*, *voxel_width: float*) → numpy.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[numpy.ndarray, numpy.ndarray]*, *atom_index_pair: Tuple[int, int]*, *box_width: float*, *voxel_width: float*) → numpy.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: Optional[Callable[[...], Any]] = None*, *feature_dict: Optional[Dict[Any, Any]] = None*, *feature_list: Optional[List[Union[int, Tuple[int]]]] = None*, *nb_channel: int = 16*, *dtype: str = 'int'*) → numpy.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

3.24.10 Graph Convolution Utilities

`one_hot_encode`(*val: Union[int, str]*, *allowable_set: Union[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: numpy.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]

3.24.11 Debug Utilities

3.24.12 Docking Utilities

These utilities assist in file preparation and processing for molecular docking.

write_vina_conf(*protein_filename: str*, *ligand_filename: str*, *centroid: numpy.ndarray*, *box_dims: numpy.ndarray*, *conf_filename: str*, *num_modes: int = 9*, *exhaustiveness: Optional[int] = 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: Optional[int] = 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: Optional[str] = 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*) → numpy.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.

3.25 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}
}
```

3.26 Contibuting to DeepChem as a Scientist

The scientific community in many ways is quite traditional. Students typically learn in apprenticeship from 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.

3.26.1 Scientist FAQ

Contents

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.

3.27 Coding Conventions

3.27.1 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.

3.27.2 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 0.22. We periodically update it to newer versions.

3.27.3 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.

3.27.4 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>
```

3.27.5 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>
```

3.27.6 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 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 env.
↪test.yml > env.yml
conda env update --file env.yml --prune
pytest -v -m 'torch' deepchem
```

3.27.7 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.

3.28 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 pe**

    - Building and installation of DeepChem in latest Ubuntu OS and Python 3.7 and it checks for `import deepchem`
    - These tests run on Ubuntu latest version using Python 3.7-3.9 and on windows latest version using Python 3.7. 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 follo**

    - For build environments of Python 3.7, 3.8 and 3.9, 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 3.7, 3.8 and 3.9 and on windows latest, it is run with Python 3.7.

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 3.7, 3.8 and 3.9.
  - 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 3.7-3.9 and on windows latest, it is run with Python 3.7.
  - 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 3.7-3.9 and on windows latest, it is run with Python 3.7.

4. **Tests for documents**
    - These tests are used for checking docs build. It is run on ubuntu latest with Python 3.7.
5. **Tests for Release**
    - These tests are run only when pushing a tag. It is run on ubuntu latest with Python 3.7.

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)
```

3.28.1 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.

3.29 Infrastructures

The DeepChem project maintains supporting infrastructure on a number of different services. This infrastructure is maintained by the DeepChem development team.

3.29.1 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.

3.29.2 Conda Forge

The DeepChem feedstock repo maintains the build recipe for conda-forge.

3.29.3 Docker Hub

DeepChem hosts major releases and nightly docker build instances on Docker Hub.

3.29.4 PyPI

DeepChem hosts major releases and nightly builds on PyPI.

3.29.5 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).

3.29.6 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.

3.29.7 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.

INDEX