DeepRank Documentation

Release 0.2

Nicolas Renaud

Feb 15, 2023

Contents

1 Tutorial **3**
- 1.1 Tutorial . . . 3
  - 1.1.1 Installation . . . 3
  - 1.1.2 Data Generation . . . 3
  - 1.1.3 Learning . . . 6
  - 1.1.4 Advanced Tutorial . . . 9

2 API Reference **13**
- 2.1 API Reference . . . 13
  - 2.1.1 Data Generation . . . 13
  - 2.1.2 Learning . . . 26
  - 2.1.3 Features . . . 40
  - 2.1.4 Targets . . . 47
  - 2.1.5 Tools . . . 49

3 Indices **53**

Python Module Index **55**

Index **57**

DeepRank is a general, configurable deep learning framework for data mining protein-protein interactions (PPIs) using 3D convolutional neural networks (CNNs).

DeepRank contains useful APIs for pre-processing PPIs data, computing features and targets, as well as training and testing CNN models.

DeepRank highlights:

- **Predefined atom-level and residue-level PPI feature types**
  - *e.g. atomic density, vdw energy, residue contacts, PSSM, etc.*
- **Predefined target types**
  - *e.g. binary class, CAPRI categories, DockQ, RMSD, FNAT, etc.*
- Flexible definition of both new features and targets
- 3D grid feature mapping
- Efficient data storage in HDF5 format
- Support both classification and regression (based on PyTorch)

CHAPTER 1

Tutorial

1.1 Tutorial

1.1.1 Installation

Installing via pypi

To install DeepRank package via pypi:

```
pip install deeprank
```

Installing from source code

To install DeepRank using GitHub source code:

```
# clone the repository
git clone https://github.com/DeepRank/deeprank.git

# enter the deeprank directory
cd deeprank

# install the module
pip install -e ./
```

1.1.2 Data Generation

This section describes how to generate feature and target data using PDB files and/or other relevant raw data, which will be directly fed to the learning step.

The generated data is stored in a single HDF5 file. The use of HDF5 format for data storage not only allows to save disk space through compression but also reduce I/O time during the deep learning step.

The following tutorial uses the example from the test file `test/test_generate.py`. All the required data, e.g. PDB files and PSSM files, can be found in the `test/1AK4/` directory. It's assumed that the working directory is `test/`.

Let's start:

First import the necessary modules,

```
>>> from mpi4py import MPI
>>> from deeprank.generate import *
```

The second line imports all the submodules required for data generation.

Then, we need to set the MPI communicator, that will be used later

```
>>> comm = MPI.COMM_WORLD
```

Calculating features and targets

The data generation requires the PDB files of docking decoys for which we want to compute features and targets. We use `pdb_source` to specify where these decoy files are located,

```
>>> pdb_source = ['./1AK4/decoys']
```

which contains 5 docking decoys of the 1AK4 complex. The structure information of these 5 decoys will be copied to the output HDF5 file.

We also need to specify the PDB files of native structures, which are required to calculate targets like RMSD, FNAT, etc.

```
>>> pdb_native = ['./1AK4/native']
```

DeepRank will automatically look for native structure for each docking decoy by name matching. For example, the native structure for the decoy `./1AK4/decoys/1AK4_100w.pdb` will be `./1AK4/native/1AK4.pdb`.

Then, if you want to compute PSSM-related features like `PSSM_IC`, you must specify the path to the PSSM files. The PSSM file must be named as `<PDB_ID>.<Chain_ID>.pssm`. To produce PSSM files that are coherent with your pdb files and have already the correct file names, you can check out our module PSSMGen.

```
>>> pssm_source = ['./1AK4/pssm_new/']
```

Finally we must specify the name of the output HDF5 file,

```
>>> h5out = '1ak4.hdf5'
```

We are now ready to initialize the `DataGenerator` class,

```
>>> database = DataGenerator(pdb_source=pdb_source,
                             pdb_native=pdb_native,
                             pssm_source=pssm_source,
                             chain1='C',
                             chain2='D',
                             compute_features=['deeprank.features.AtomicFeature',
                                               'deeprank.features.FullPSSM',
                                               'deeprank.features.PSSM_IC',
                                               'deeprank.features.BSA'],
                             compute_targets=['deeprank.targets.dockQ'],
                             hdf5=h5out)
```

The `compute_features` and `compute_targets` are used to set which features and targets we want to calculate by providing a list of python class names. The available predefined features and targets can be found in the API Reference.

The above statement initializes a class instance but does not compute anything yet. To actually execute the calculation, we must use the `create_database()` method,

```
>>> database.create_database(prog_bar=True)
```

Once you punch that DeepRank will fo through all the protein complexes specified as input and compute all the features and targets required.

Mapping features to 3D grid

The next step consists in mapping the features calculated above to a grid of points centered around the molecule interface. Before mapping the features, we must define the grid first,

```
>>> grid_info = {'number_of_points': [30, 30, 30],
                 'resolution': [1., 1., 1.],
                 'atomic_densities': {'C': 1.7, 'N': 1.55, 'O': 1.52, 'S': 1.8}
                }
```

Here we define a grid with 30 points in x/y/z direction and a distance interval of 1Å between two neighbor points. We also provide the atomic radius for the element *C*, *N*, *O* and *S* to calculate atomic densities. Atomic density is also a type of features which have to be calculated during the mapping.

Then we start mapping the features to 3D grid,

```
>>> database.map_features(grid_info, try_sparse=True, prog_bar=True)
```

By setting `try_sparse` to `True`, DeepRank will try to store the 3D grids with a built-in sparse format, which can seriously save the storage space.

Finally, it should generate the HDF5 file `1ak4.hdf5` that contains all the raw features and targets data and the grid-mapped data which will be used for the learning step. To easily explore these data, you could try the DeepXplorer tool.

Appending features/targets

Suppose you've finished generating a huge HDF5 database and just realize you forgot to compute some specific features or targets. Do you have to recompute everything?

No! You can append more features and targets to the existing HDF5 database in a very simple way:

```
>>> h5file = '1ak4.hdf5'
>>> database = DataGenerator(compute_targets=['deeprank.targets.binary_class'],
>>>                          compute_features=['deeprank.features.ResidueDensity'],
>>>                          hdf5=h5file)
>>>
>>> # add targets
>>> database.add_target()
>>>
>>> # add features
>>> database.add_feature()
>>>
>>> # map features
>>> database.map_features()
```

Voilà! Here we simply specify the name of the existing HDF5 file we generated above, and set the new features/targets to add to this database. The methods `add_target` and `add_feature` are then called to calculate the corresponding targets and features. Don't forget to map the new features afterwards. Note that you don't have to provide any grid information for the mapping, because DeepRank will automatically detect and use the grid info that exist in the HDF5 file.

1.1.3 Learning

This section describes how to prepare training/validation/test datasets with specific features and targets, how to configure neural network architecture, and how to train and test a neural network with DeepRank.

This tutorial uses the example from the test file `test/test_learn.py`.

Let's start from importing the necessary DeepRank modules,

```
>>> from deeprank.learn import Dataset, NeuralNet
```

The `Dataset` module is used for preparing training/validation/test datasets and the `NeuralNet` module for training and testing neural networks.

Creating training/validation/test datasets

First we need to provide the HDF5 file we generated in the Data Generation step,

```
>>> database = '1ak4.hdf5'
```

You can also provide multiple HDF5 files to a database, e.g.

```
>>> database = ['1ak4.hdf5','native.hdf5']
```

Then we will use the data from this `database` to create the training/validation/test datasets,

```
>>> data_set = DataSet(train_database=database,
>>>                    valid_database=None,
>>>                    test_database=None,
>>>                    chain1='C',
>>>                    chain2='D',
>>>                    grid_shape=(30, 30, 30),
>>>                    select_feature={
>>>                      'AtomicDensities_ind': 'all',
>>>                      'Feature_ind': ['coulomb', 'vdwaals', 'charge','PSSM_*']},
>>>                     select_target='BIN_CLASS',
>>>                     normalize_features=True,
>>>                     normalize_targets=False,
>>>                     pair_chain_feature=np.add,
>>>                     dict_filter={'IRMSD':'<4. or >10.'})
```

Here we have only one database, so we must provide it to the `train_database` parameter, and later you will set how to split this database to training, validation and test data. When independent validation and test databases exist, you can then set the corresponding `valid_database` and `test_database` parameters.

You must also specify which features and targets to use for the neural network by setting the `select_feature` and `select_target` parameters. By default, all features exist in the HDF5 files will be selected.

When you specify some of the features, you have to use a *dictionary*. As you can see in the HDF5 file, the `mapped_features` subgroup of each complex contains two groups `AtomicDensities_ind` and

`Feature_ind`. We'll forget about the `_ind` that is for legacy reasons, but these two groups contains the atomic densities and other features respectively. Therefore the value used in the example above:

```
>>> select_feature={'AtomicDensities': 'all',
>>>                 'Features': ['coulomb', 'vdwaals', 'charge', 'PSSM_*']}
```

It means that we want to use all the atomic densities features, the `coulomb`, `vdwaals` and `charge` features as well as the PSSM features with a name starting with `PSSM_`.

With the `normalize_features` and `normalize_targets` parameters, you can use the normalized values of features or targets to train a neural network.

As you can see in the HDF5 file for each feature, the data of `chain1` and `chain2` are stored separately. But it is possible to combine the feature of both chains into a single channel via the parameter `pair_chain_feature`, e.g.,

```
>>> pair_chain_feature = np.add
```

which means that feature value of `chain1` and `chain2` will be summed up to create a single channel.

You can further filter the data with specific conditions,

```
>>> dict_filter={'IRMSD':'<4. or >10.'}
```

it will only selects the complexes whose IRMSD are lower than 4Å or larger than 10Å.

Creating neural network architecture

The architecture of the neural network has to be well defined before training and DeepRank provides a very useful tool `modelGenerator` to facilitate this process.

Here is a simple example showing how to generate NN architecture:

```
>>> from deeprank.learn.modelGenerator import *
>>>
>>> conv_layers = []
>>> conv_layers.append(conv(output_size=4,kernel_size=2,post='relu'))
>>> conv_layers.append(pool(kernel_size=2))
>>> conv_layers.append(conv(input_size=4,output_size=5,kernel_size=2,post='relu'))
>>> conv_layers.append(pool(kernel_size=2))
>>>
>>> fc_layers = []
>>> fc_layers.append(fc(output_size=84,post='relu'))
>>> fc_layers.append(fc(input_size=84,output_size=2))
>>>
>>> gen = NetworkGenerator(name='cnn_demo',
>>>                        fname='cnn_arch_demo.py',
>>>                        conv_layers=conv_layers,
>>>                        fc_layers=fc_layers)
>>> gen.print()
>>> gen.write()
```

It will print out the human readable summary of the architecture,

```
#------------------------------------------------------------------------
# Network Structure
#------------------------------------------------------------------------
#conv layer   0: conv | input -1  output  4  kernel  2  post relu
#conv layer   1: pool | kernel  2  post None
```

(continues on next page)

(continued from previous page)

```
#conv layer   2: conv | input  4  output  5  kernel  2  post relu
#conv layer   3: pool | kernel  2  post None
#fc   layer   0: fc   | input -1  output  84  post relu
#fc   layer   1: fc   | input  84  output  2  post None
#-------------------------------------------------------------------
```

and also generate a `cnn_arch_demo.py` file with a class `cnn_demo` that defines the NN architecture.

We also provide the predefined NN architectures in `learn/model3d.py` and `learn/model2d.py`.

Training a neural network

We are now all set to start the deep learning experiments. DeepRank supports both classification and regression tasks.

Classification with 3D CNN

```
>>> from deeprank.learn.model3d import cnn_class
>>>
>>> model = NeuralNet(data_set=data_set,
>>>                   model=cnn_class,
>>>                   model_type='3d',
>>>                   task='class')
```

`data_set` is the dataset created above and `cnn_class` is the predefined NN architecture. We also need to specify the `model_type` and the learning `task`.

Then we can start the training process,

```
>>> model.train(nepoch=50,
>>>             divide_trainset=[0.7, 0.2, 0.1]
>>>             train_batch_size=5,
>>>             num_workers=1,
>>>             hdf5='epoch_data_class.hdf5')
```

We specify here the number of epoch, the fraction of data for training/validation/test sets, the batch size and the number of workers (CPU threads) in charge of batch preparation, and the output HDF5 file for training results. The model will be saved to `.pth.tar` files, e.g. `model_epoch_0001.pth.tar`.

Regression with 3D CNN

To train a regression model, the steps are same as the classification above. But you need to provide the regression NN architecture and set the correct task type, e.g.

```
>>> from deeprank.learn.model3d import cnn_reg
>>>
>>> model = NeuralNet(data_set=data_set,
>>>                   model=cnn_reg,
>>>                   model_type='3d',
>>>                   task='reg')
>>>
>>> model.train(nepoch=50,
>>>             divide_trainset=[0.7, 0.2, 0.1]
```

(continues on next page)

(continued from previous page)

```
>>>             train_batch_size=5,
>>>             num_workers=1,
>>>             hdf5='epoch_data_reg.hdf5')
```

2D CNN

DeepRank allows to transform the 3D volumetric data to 2D data by slicing planes of the data and using each plane as given channel. Very little modification of the code are necessary to do so. The creation of the dataset is identical to the 3D case, and you must specify `model_type=2d` for `NeuralNet`,

```
>>> from deeprank.learn.model2d import cnn
>>>
>>> model = NeuralNet(data_set=data_set_2d,
>>>                   model=cnn,
>>>                   model_type='2d',
>>>                   task='reg',
>>>                   proj2d=0)
```

The `proj2d` parameter defines how to slice the 3D volumetric data. Value of: 0, 1, 2 are possible to slice along the YZ, XZ or XY plane respectively.

Testing a neural network

In many cases after you've trained the NN model, you would like to use the model to do prediction or to test the model's performance on new data. DeepRank provide a very easy way to do that, let's say we have got the trained classification model `model.pth.tar`,

```
>>> from deeprank.learn.model3d import cnn_class
>>>
>>> database = '1AK4_test.hdf5'
>>> model = NeuralNet(database, cnn_class, pretrained_model='model.pth.tar')
>>> model.test()
```

Note that here the database is simply the name of the hdf5 file we want to test the model on. All the processing of the dataset will be automatically done in the exact same way as it was done during the training of the model. Hence you do not have to copy the `select_features` and `select_target` parameters, all that is done for you automatically.

1.1.4 Advanced Tutorial

DeepRank allows users to define and create new features and targets, and this section will show how to that.

Create your own features

To create your own feature you simply have to create a feature class that must subclass the `FeatureClass` contained in `features/FeatureClass.py`. As an example we will create here a new feature that maps the carbon alpha of the contact residue. The first thing we need to do is to import `pdb2sql` and the `FeatureClass` superclass,

```
>>> import pdb2sql
>>> from deeprank.features import FeatureClass
```

We then have to define the class and its initialization. Here we will simply initialize the class with the pdb information of the molecule we want to process. This is therefore given by

```
>>> # a new class based on the FeatureClass
>>> class CarbonAlphaFeature(FeatureClass):
>>>
>>>     # init the class
>>>     def __init__(self,pdbfile):
>>>         super().__init__('Atomic')
>>>         self.pdb = pdb
>>>
>>>     # the feature extractor
>>>     def get_feature(self):
>>>
>>>         # create a sql database
>>>         db = pdb2sql(self.pdb)
>>>
>>>         # get the contact atoms
>>>         indA,indB = db.get_contact_atoms()
>>>         contact = indA + indB
>>>
>>>         # extract the atom keys and xyz of the contact CA atoms
>>>         ca_keys = db.get('chainID,resName,resSeq,name',name='CA',rowID=contact)
>>>         ca_xyz = db.get('x,y,z',name='CA',rowID=contact)
>>>
>>>         # create the dictionary of human readable and xyz-val data
>>>         hread, xyzval = {},{}
>>>         for key,xyz in zip(ca_keys,ca_xyz):
>>>
>>>             # human readable
>>>             # { (chainID,resName,resSeq,name): [val] }
>>>             hread[tuple(key)] = [1.0]
>>>
>>>             # xyz-val
>>>             # { (0|1,x,y,z): [val] }
>>>             chain = [{'A':0,'B':1}[key[0]]]
>>>             k = tuple( chain + xyz)
>>>             xyzval[k] = [1.0]
>>>
>>>         self.feature_data['CA'] = hread
>>>         self.feature_data_xyz['CA'] = xyzval
```

As you can see we must initialize the superclass. Since we use here the argument `Atomic`, the feature is an atomic based feature. If we would create a residue based feature (e.g. PSSM, residue contact, ...) we would have used here the argument `Residue`. This argument specifies the printing format of the feature in a human readable format and doesn't affect the mapping. From the super class the new class inherit two methods

```
>>> export_data_hdf5(self,featgrp)
>>> export_dataxyz_hdf5(self,featgrp)
```

which are used to store feature values in the HDF5 file.

The class also inherits two variables

```
>>> self.feature_data = {}
>>> self.feature_data_xyz = {}
```

where we must store the feature in human readable format and xyz-val format.

To extract the feature value we are going to write a method in the class in charge of the feature extraction. This method is simply going to locate the carbon alpha and gives a value of 1 at the corresponding xyz positions.

In this function we exploit `pdb2sql` to locate the carbon alpha that make a contact. We then create two dictionaries where we store the feature value. The format of the human readable and xyz-val are given in comment. These two dictionaries are then added to the superclass variable `feature_data` and `feature_data_xyz`.

We now must use this new class that we've just created in DeepRank. To do that we must create a function called:

```
>>> def __compute_feature__(pdb_data,featgrp,featgrp_raw)
```

Several example can be found in the feature already included in DeepRank. The location of the this function doesn't matter as we will provide the python file as value of the `compute_features` argument in the `DataGenerator`. For the feature we have just created we can define that function as

```
>>> def __compute_feature__(pdb_data,featgrp,featgrp_raw):
>>>
>>>         cafeat = CarbonAlphaFeature(pdb_data)
>>>         cafeat.get_features()
>>>
>>>         # export in the hdf5 file
>>>         cafeat.export_dataxyz_hdf5(featgrp)
>>>         cafeat.export_data_hdf5(featgrp_raw)
>>>
>>>         # close
>>>         cafeat.db._close()
```

Finally to compute this feature we must call it during the data generation process. Let's assume that the file containing the `__compute_feature__` function is in the local folder and is called `CAfeature.py`. To use this new feature in the generation we can simply pass the name of this file in the DataGenerator as

```
>>> database = DataGenerator(pdb_source=pdb_source, pdb_native=pdb_native,
>>>                          compute_features = ['CAFeature', ....], ...)
```

Create your own targets

The creation of new targets is similar to the process of creating new features but simpler. The targets don't need to be mapped on a grid and therefore don't need any fancy formatting. We simply need to create a new dataset in the target group of the molecule concerned. For example let's say we want to associate a random number to each conformation. To do that we can use the following code:

```
>>> import numpy as np
>>>
>>> def get_random_number():
>>>     return np.random.rand()
>>>
>>> def __compute_target__(pdb_data, targrp):
>>>
>>>     target = get_random_number()
>>>     targrp.create_dataset('FOO', data=np.array(target))
```

As for the features, the new target must be called in a function with a very precise name convention:

```
>>> def __compute_target__(pdb_data, targrp)
```

If as before we assume that the file containing this function is in the local folder and is called `random.py` we can compute the target by calling the `DataGenerator` with:

```
>>> database = DataGenerator(pdb_source=pdb_source, pdb_native=pdb_native,
>>>                          compute_targets = ['random',....], ...)
```

CHAPTER 2

API Reference

2.1 API Reference

2.1.1 Data Generation

This module contains all the tools to compute the features and targets and to map the features onto a grid of points. The main class used for the data generation is `deeprank.generate.DataGenerator`. Through this class you can specify the molecules you want to consider, the features and the targets that need to be computed and the way to map the features on the grid. The data is stored in a single HDF5 file. In this file, each conformation has its own group that contains all the information related to the conformation. This includes the pdb data, the value of the feature (in human readable format and xyz-val format), the value of the targe values, the grid points and the mapped features on the grid.

At the moment a number of features are already implemented. This include:

- Atomic densities
- Coulomb & vd Waals interactions
- Atomic charges
- PSSM data
- Information content
- Buried surface area
- Contact Residue Densities

More features can be easily implemented and integrated in the data generation workflow. You can see example here. The calculation of a number of target values have also been implemented:

- i-RMSD
- l-RMSD
- FNAT

- DockQ
- binary class

There as well new targets can be implemented and integrated to the workflow.

Normalization of the data can be time consuming as the dataset becomes large. As an attempt to alleviate this problem, the class `deeprank.generate.NormalizeData` has been created. This class directly compute and store the standard deviation and mean value of each feature within a given hdf5 file.

Example:

```
>>> from deeprank.generate import *
>>> from time import time
>>>
>>> pdb_source     = ['./1AK4/decoys/']
>>> pdb_native     = ['./1AK4/native/']
>>> pssm_source    = './1AK4/pssm_new/'
>>> h5file = '1ak4.hdf5'
>>>
>>> #init the data assembler
>>> database = DataGenerator(chain1='C',chain2='D',
>>>                          pdb_source=pdb_source,pdb_native=pdb_native,pssm_
↪source=pssm_source,
>>>                          data_augmentation=None,
>>>                          compute_targets  = ['deeprank.targets.dockQ'],
>>>                          compute_features = ['deeprank.features.AtomicFeature',
>>>                                              'deeprank.features.FullPSSM',
>>>                                              'deeprank.features.PSSM_IC',
>>>                                              'deeprank.features.BSA'],
>>>                          hdf5=h5file)
>>>
>>> t0 = time()
>>> #create new files
>>> database.create_database(prog_bar=True)
>>>
>>> # map the features
>>> grid_info = {
>>>     'number_of_points': [30,30,30],
>>>     'resolution': [1.,1.,1.],
>>>     'atomic_densities': {'CA':3.5,'N':3.5,'O':3.5,'C':3.5},
>>> }
>>> database.map_features(grid_info,try_sparse=True,time=False,prog_bar=True)
>>>
>>> # add a new target
>>> database.add_target(prog_bar=True)
>>> print(' '*25 + '--> Done in %f s.' %(time()-t0))
>>>
>>> # get the normalization
>>> norm = NormalizeData(h5file)
>>> norm.get()
```

The details of the different submodule are listed here. The only module that really needs to be used is `DataGenerator` and `NormalizeData`. The `GridTools` class should not be directly used by inexperienced users.

Structure Alignement

All the complexes contained in the dataset can be aligned similarly to facilitate and improve the training of the model. This can easily be done using the *align* option of the *DataGenerator* for example to align all the complexes along the 'z' direction one can use:

```
>>> database = DataGenerator(chain1='C',chain2='D',
>>>                          pdb_source=pdb_source, pdb_native=pdb_native, pssm_
↪source=pssm_source,
>>>                          align={"axis":'z'}, data_augmentation=2,
>>>                          compute_targets=[ ... ], compute_features=[ ... ], ... )
```

Other options are possbile, for example if you would like to have the alignement done only using a subpart of the complex, say the chains A and B you can use:

```
>>> database = DataGenerator(chain1='C',chain2='D',
>>>                          pdb_source=pdb_source, pdb_native=pdb_native, pssm_
↪source=pssm_source,
>>>                          align={"axis":'z', "selection": {"chainID":["A","B"]} },␣
↪data_augmentation=2,
>>>                          compute_targets=[ ... ], compute_features=[ ... ], ... )
```

All the selection offered by *pdb2sql* can be used in the *align* dictionnary e.g.: "resId":[1,2,3], "resName":['VAL','LEU'], ... Only the atoms selected will be aligned in the give direction.

You can also try to align the interface between two chains in a given plane. This can be done using:

```
>>> database = DataGenerator(chain1='C',chain2='D',
>>>                          pdb_source=pdb_source, pdb_native=pdb_native, pssm_
↪source=pssm_source,
>>>                          align={"plane":'xy', "selection":"interface"}, data_
↪augmentation=2,
>>>                          compute_targets=[ ... ], compute_features=[ ... ], ... )
```

which by default will use the interface between the first two chains. If you have more than two chains in the complex and want to specify which chains are forming the interface to be aligned you can use:

```
>>> database = DataGenerator(chain1='C',chain2='D',
>>>                          pdb_source=pdb_source, pdb_native=pdb_native, pssm_
↪source=pssm_source,
>>>                          align={"plane":'xy', "selection":"interface", "chain1":'A
↪', "chain2":'C'}, data_augmentation=2,
>>>                          compute_targets=[ ... ], compute_features=[ ... ], ... )
```

DataGenerator

deeprank.generate.DataGenerator.**_printif**(*string*, *cond*)

class deeprank.generate.DataGenerator.**DataGenerator**(*chain1*, *chain2*, *pdb_select=None*, *pdb_source=None*, *pdb_native=None*, *pssm_source=None*, *align=None*, *compute_targets=None*, *compute_features=None*, *data_augmentation=None*, *hdf5='database.h5'*, *mpi_comm=None*)

Bases: object

Generate the data (features/targets/maps) required for deeprank.

Parameters

- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID
- **pdb_select** (*list(str), optional*) – List of individual conformation for mapping
- **pdb_source** (*list(str), optional*) – List of folders where to find the pdbs for mapping
- **pdb_native** (*list(str), optional*) – List of folders where to find the native conformations, nust set it if having targets to compute in parameter "compute_targets".
- **pssm_source** (*list(str), optional*) – List of folders where to find the PSSM files
- **align** (*dict, optional*) – Dicitionary to align the compexes, e.g. align = {"selection":{"chainID":["A","B"]},"axis":"z"}} e.g. align = {"selection":"interface","plane":"xy"} if "selection" is not specified the entire complex is used for alignement
- **compute_targets** (*list(str), optional*) – List of python files computing the targets, "pdb_native" must be set if having targets to compute.
- **compute_features** (*list(str), optional*) – List of python files computing the features
- **data_augmentation** (*int, optional*) – Number of rotation performed one each complex
- **hdf5** (*str, optional*) – name of the hdf5 file where the data is saved, default to 'database.h5'
- **mpi_comm** (*MPI_COMM*) – MPI COMMUNICATOR

Raises NotADirectoryError – if the source are not found

Example

```
>>> from deeprank.generate import *
>>> # sources to assemble the data base
>>> pdb_source     = ['./1AK4/decoys/']
>>> pdb_native     = ['./1AK4/native/']
```

(continues on next page)

(continued from previous page)

```
>>> pssm_source    = ['./1AK4/pssm_new/']
>>> h5file = '1ak4.hdf5'
>>>
>>> #init the data assembler
>>> database = DataGenerator(chain1='C',
>>>                          chain2='D',
>>>                          pdb_source=pdb_source,
>>>                          pdb_native=pdb_native,
>>>                          pssm_source=pssm_source,
>>>                          data_augmentation=None,
>>>                          compute_targets=['deeprank.targets.dockQ'],
>>>                          compute_features=['deeprank.features.AtomicFeature',
>>>                                            'deeprank.features.PSSM_IC',
>>>                                            'deeprank.features.BSA'],
>>>                          hdf5=h5file)
```

create_database(*verbose=False*, *remove_error=True*, *prog_bar=False*, *contact_distance=8.5*, *random_seed=None*)

Create the hdf5 file architecture and compute the features/targets.

Parameters

- **verbose** (*bool, optional*) – Print creation details
- **remove_error** (*bool, optional*) – remove the groups that errored
- **prog_bar** (*bool, optional*) – use tqdm
- **contact_distance** (*float*) – contact distance cutoff, defaults to 8.5Å
- **random_seed** (*int*) – random seed for getting rotation axis and angle

Raises `ValueError` – If creation of the group errored.

Example:

```
>>> # sources to assemble the data base
>>> pdb_source     = ['./1AK4/decoys/']
>>> pdb_native     = ['./1AK4/native/']
>>> pssm_source    = ['./1AK4/pssm_new/']
>>> h5file = '1ak4.hdf5'
>>>
>>> #init the data assembler
>>> database = DataGenerator(chain1='C',
>>>                          chain2='D',
>>>                          pdb_source=pdb_source,
>>>                          pdb_native=pdb_native,
>>>                          pssm_source=pssm_source,
>>>                          data_augmentation=None,
>>>                          compute_targets  = ['deeprank.targets.dockQ'],
>>>                          compute_features = ['deeprank.features.
 ↪AtomicFeature',
>>>                                              'deeprank.features.PSSM_IC',
>>>                                              'deeprank.features.BSA'],
>>>                          hdf5=h5file)
>>>
>>> #create new files
>>> database.create_database(prog_bar=True)
```

aug_data (*augmentation*, *keep_existing_aug=True*, *random_seed=None*)
Augment exiting original PDB data and features.

Parameters

- **augmentation** (*int*) – Times of augmentation
- **keep_existing_aug** (*bool, optional*) – Keep existing augmentated data. If False, existing aug will be removed. Defaults to True.

Examples

```
>>> database = DataGenerator(h5='database.h5')
>>> database.aug_data(augmentation=3, append=True)
>>> grid_info = {
>>>     'number_of_points': [30,30,30],
>>>     'resolution': [1.,1.,1.],
>>>     'atomic_densities': {'C':1.7, 'N':1.55, 'O':1.52, 'S':1.8},
>>>     }
>>> database.map_features(grid_info)
```

add_feature (*remove_error=True*, *prog_bar=True*)
Add a feature to an existing hdf5 file.

Parameters

- **remove_error** (*bool*) – remove errored molecule
- **prog_bar** (*bool, optional*) – use tqdm

Example:

```
>>> h5file = '1ak4.hdf5'
>>>
>>> #init the data assembler
>>> database = DataGenerator(compute_features  = ['deeprank.features.
↪ResidueDensity'],
>>>                          hdf5=h5file)
>>>
>>> database.add_feature(remove_error=True, prog_bar=True)
```

add_unique_target (*targdict*)
Add identical targets for all the complexes in the datafile.

This is usefull if you want to add the binary class of all the complexes created from decoys or natives

Parameters **targdict** (*dict*) – Example: {'DOCKQ':1.0}

```
>>> database = DataGenerator(hdf5='1ak4.hdf5')
>>> database.add_unique_target({'DOCKQ':1.0})
```

add_target (*prog_bar=False*)
Add a target to an existing hdf5 file.

Parameters **prog_bar** (*bool, optional*) – Use tqdm

Example:

```
>>> h5file = '1ak4.hdf5'
>>>
>>> #init the data assembler
>>> database = DataGenerator(compute_targets =['deeprank.targets.binary_class
↪'],
>>>                          hdf5=h5file)
>>>
>>> database.add_target(prog_bar=True)
```

realign_complexes(*align*, *compute_features=None*, *pssm_source=None*)

Align all the complexes already present in the HDF5.

Parameters **{dict} -- alignement dictionary** (*align*) –

Keyword Arguments **{list} -- list of features to be computed** (*compute_features*) –

if None computes the features specified in the attrs['features'] of the file (if present)

pssm_source {str} – path of the pssm files. If None the source specfied in the attrs['pssm_source'] will be used (if present) (default: {None})

Raises ValueError – If no PSSM detected

Example:

```
>>> database = DataGenerator(hdf5='1ak4.hdf5')
>>> # if comute_features and pssm_source are not specified
>>> # the values in hdf5.attrs['features'] and hdf5.attrs['pssm_source'] will␣
↪be used
>>> database.realign_complex(align={'axis':'x'},
>>>                          compute_features['deeprank.features.X'],
>>>                           pssm_source='./1ak4_pssm/')
```

_get_grid_center(*pdb*, *contact_distance*)

precompute_grid(*grid_info*, *contact_distance=8.5*, *prog_bar=False*, *time=False*, *try_sparse=True*)

map_features(*grid_info={}*, *cuda=False*, *gpu_block=None*, *cuda_kernel='kernel_map.c'*, *cuda_func_name='gaussian'*, *try_sparse=True*, *reset=False*, *use_tmpdir=False*, *time=False*, *prog_bar=True*, *grid_prog_bar=False*, *remove_error=True*)

Map the feature on a grid of points centered at the interface.

If features to map are not given, they will be are automatically determined for each molecule. Otherwise, given features will be mapped for all molecules (i.e. existing mapped features will be recalculated).

Parameters

- **grid_info** (*dict*) – Informaton for the grid. See deeprank.generate.GridTools.py for details.
- **cuda** (*bool, optional*) – Use CUDA
- **gpu_block** (*None, optional*) – GPU block size to be used
- **cuda_kernel** (*str, optional*) – filename containing CUDA kernel
- **cuda_func_name** (*str, optional*) – The name of the function in the kernel
- **try_sparse** (*bool, optional*) – Try to save the grids as sparse format
- **reset** (*bool, optional*) – remove grids if some are already present
- **use_tmpdir** (*bool, optional*) – use a scratch directory

- **time** (*bool, optional*) – time the mapping process
- **prog_bar** (*bool, optional*) – use tqdm for each molecule
- **grid_prog_bar** (*bool, optional*) – use tqdm for each grid
- **remove_error** (*bool, optional*) – remove the data that errored

Example:

```
>>> #init the data assembler
>>> database = DataGenerator(hdf5='1ak4.hdf5')
>>>
>>> # map the features
>>> grid_info = {
>>>     'number_of_points': [30,30,30],
>>>     'resolution': [1.,1.,1.],
>>>     'atomic_densities': {'C':1.7, 'N':1.55, 'O':1.52, 'S':1.8},
>>> }
>>>
>>> database.map_features(grid_info,try_sparse=True,time=False,prog_bar=True)
```

remove (*feature=True*, *pdb=True*, *points=True*, *grid=False*)
Remove data from the data set.

Equivalent to the cleandata command line tool. Once the data has been removed from the file it is impossible to add new features/targets

Parameters

- **feature** (*bool, optional*) – Remove the features
- **pdb** (*bool, optional*) – Remove the pdbs
- **points** (*bool, optional*) – remove teh grid points
- **grid** (*bool, optional*) – remove the maps

_tune_cuda_kernel (*grid_info*, *cuda_kernel='kernel_map.c'*, *func='gaussian'*)
Tune the CUDA kernel using the kernel tuner http://benvanwerkhoven.github.io/kernel_tuner/

Parameters

- **grid_info** (*dict*) – information for the grid definition
- **cuda_kernel** (*str, optional*) – file containing the kernel
- **func** (*str, optional*) – function in the kernel to be used

Raises ValueError – If the tuner has not been used

_test_cuda (*grid_info*, *gpu_block=8*, *cuda_kernel='kernel_map.c'*, *func='gaussian'*)
Test the CUDA kernel.

Parameters

- **grid_info** (*dict*) – Information for the grid definition
- **gpu_block** (*int, optional*) – GPU block size to be used
- **cuda_kernel** (*str, optional*) – File containing the kernel
- **func** (*str, optional*) – function in the kernel to be used

Raises ValueError – If the kernel has not been installed

static _compile_cuda_kernel(*cuda_kernel*, *npts*, *res*)
Compile the cuda kernel.

Parameters

- **cuda_kernel** (*str*) – filename
- **npts** (*tuple(int)*) – number of grid points in each direction
- **res** (*tuple(float)*) – resolution in each direction

Returns compiled kernel

Return type compiler.SourceModule

static _get_cuda_function(*module*, *func_name*)
Get a single function from the compiled kernel.

Parameters

- **module** (*compiler.SourceModule*) – compiled kernel module
- **func_name** (*str*) – Name of the funtion

Returns cuda function

Return type func

static _tunable_kernel(*kernel*)
Make a tunale kernel.

Parameters **kernel** (*str*) – String of the kernel

Returns tunable kernel

Return type TYPE

_filter_cplx()
Filter the name of the complexes.

static _compute_features(*feat_list*, *pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*, *logger*)
Compute the features.

Parameters

- **feat_list** (*list(str)*) – list of function name, e.g., ['deeprank.features.ResidueDensity', 'deeprank.features.PSSM_IC']
- **pdb_data** (*bytes*) – PDB translated in bytes
- **featgrp** (*str*) – name of the group where to store the xyz feature
- **featgrp_raw** (*str*) – name of the group where to store the raw feature
- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID
- **logger** (*logger*) – name of logger object

Returns error happened or not

Return type bool

static _compute_targets(*targ_list*, *pdb_data*, *targrp*)
Compute the targets.

Parameters

- **targ_list** (*list(str)*) – list of function name
- **pdb_data** (*bytes*) – PDB translated in btes
- **targrp** (*str*) – name of the group where to store the targets
- **logger** (*logger*) – name of logger object

_add_pdb(*molgrp*, *pdbfile*, *name*)
Add a pdb to a molgrp.

Parameters

- **molgrp** (*str*) – mopl group where tp add the pdb
- **pdbfile** (*str*) – psb file to add
- **name** (*str*) – dataset name in the hdf5 molgroup

_get_aligned_sqldb(*pdbfile*, *dict_align*)
return a sqldb of the pdb that is aligned as specified in the dict

Parameters

- **{str} -- path ot the pdb** (*pdbfile*) –
- **{dict} -- dictionanry of options to align the pdb** (*dict_align*) –

static _get_aligned_rotation_axis_angle(*random_seed*, *dict_align*)

Returns the axis and angle of rotation for data augmentation with aligned complexes

Parameters

- **{int} -- random seed of rotation** (*random_seed*) –
- **{dict} -- the dict describing the alignement** (*dict_align*) –

Returns axis of rotation float: angle of rotation

Return type list(float)

_add_aug_pdb(*molgrp*, *pdbfile*, *name*, *axis*, *angle*)
Add augmented pdbs to the dataset.

Parameters

- **molgrp** (*str*) – name of the molgroup
- **pdbfile** (*str*) – pdb file name
- **name** (*str*) – name of the dataset
- **axis** (*list(float)*) – axis of rotation
- **angle** (*float*) – angle of rotation
- **dict_align** (*dict*) – dict for alignement of the original pdb

Returns center of the molecule

Return type list(float)

static _rotate_feature(*molgrp*, *axis*, *angle*, *center*, *feat_name='all'*)
Rotate the raw feature values.

Parameters

- **molgrp** (*str*) – name pf the molgrp
- **axis** (*list(float)*) – axis of rotation
- **angle** (*float*) – angle of rotation
- **center** (*list(float)*) – center of rotation
- **feat_name** (*str*) – name of the feature to rotate or 'all'

NormalizeData

class deeprank.generate.NormalizeData.**NormalizeData**(*fname*, *shape=None*)

Compute the normalization factor for the features and targets of a given HDF5 file.

The normalization of the features is done through the NormParam class that assumes gaussian distribution. Hence the Normalized data should be normally distributed with a 0 mean value and 1 standard deviation. The normalization of the targets is done vian a min/max normalization. As a result the normalized targets should all lie between 0 and 1. By default the output file containing the normalization dictionary is called <hdf5name>_norm.pckl

Parameters

- **fname** (*str*) – name of the hdf5 file
- **shape** (*tuple(int), optional*) – shape of the grid in the hdf5 file

Example

```
>>> norm = NormalizeData('1ak4.hdf5')
>>> norm.get()
```

get()

Get the normalization and write them to file.

_load()

Load data from already existing normalization file.

_extract_shape()

Get the shape of the data in the hdf5 file.

_extract_data()

Extract the data from the different maps.

_process_data()

Compute the standard deviation of the data.

_export_data()

Pickle the data to file.

class deeprank.generate.NormalizeData.**NormParam**(*std=0*, *mean=0*, *var=0*, *sqmean=0*)

Compute gaussian normalization for a given feature.

This class allows to extract the standard deviation, mean value, variance and square root of the mean value of a mapped feature stored in the hdf5 file. As the entire data set is too large to fit in memory, the standard deviation of a given feature is calculated from the std of all the individual grids. This is done following: https://stats.stackexchange.com/questions/25848/how-to-sum-a-standard-deviation:

$$\sigma_{tot} = \sqrt{\frac{1}{N}\sum_i \sigma_i^2 + \frac{1}{N}\sum_i \mu_i^2 - (\frac{1}{N}\sum_i \mu_i)^2}$$

Parameters

- **std** (*float, optional*) – standard deviation
- **mean** (*float,optional*) – mean value
- **var** (*float,optional*) – variance
- **sqmean** (*float, optional*) – square roo of the variance

add(*mean*, *var*)
Add the mean value, sqmean and variance of a new molecule to the corresponding attributes.

process(*n*)
Compute the standard deviation of the ensemble.

class deeprank.generate.NormalizeData.**MinMaxParam**(*minv=None*, *maxv=None*)
Compute the min/max of an ensenble of data.

This is principally used to normalized the target values

Parameters

- **minv** (*float, optional*) – minimal value
- **maxv** (*float, optional*) – maximal value

update(*val*)

GridTools

deeprank.generate.GridTools.**logif**(*string*, *cond*)

class deeprank.generate.GridTools.**GridTools**(*molgrp*, *chain1*, *chain2*, *number_of_points=30*, *resolution=1.0*, *atomic_densities=None*, *atomic_densities_mode='ind'*, *feature=None*, *feature_mode='ind'*, *contact_distance=8.5*, *cuda=False*, *gpu_block=None*, *cuda_func=None*, *cuda_atomic=None*, *prog_bar=False*, *time=False*, *try_sparse=True*)
Map the feature of a complex on the grid.

Parameters

- **molgrp** (*str*) – name of the group of the molecule in the HDF5 file.
- **chain1** (*str*) – First chain ID.
- **chain2** (*str*) – Second chain ID.
- **number_of_points** (*int, optional*) – number of points we want in each direction of the grid.
- **resolution** (*float, optional*) – distance(in Angs) between two points.
- **atomic_densities** (*dict, optional*) – dictionary of element types with their vdw radius, see deeprank.config.atom_vdw_radius_noH
- **atomic_densities_mode** (*str, optional*) – Mode for mapping (deprecated must be 'ind').

- **feature** (*None, optional*) – Name of the features to be mapped. By default all the features present in hdf5_file['< molgrp > /features/] will be mapped.
- **feature_mode** (*str, optional*) – Mode for mapping (deprecated must be 'ind').
- **contact_distance** (*float, optional*) – the dmaximum distance between two contact atoms default 8.5Å.
- **cuda** (*bool, optional*) – Use CUDA or not.
- **gpu_block** (*tuple(int), optional*) – GPU block size to use.
- **cuda_func** (*None, optional*) – Name of the CUDA function to be used for the mapping of the features. Must be present in kernel_cuda.c.
- **cuda_atomic** (*None, optional*) – Name of the CUDA function to be used for the mapping of the atomic densities. Must be present in kernel_cuda.c.
- **prog_bar** (*bool, optional*) – print progression bar for individual grid (default False).
- **time** (*bool, optional*) – print timing statistic for individual grid (default False).
- **try_sparse** (*bool, optional*) – Try to store the matrix in sparse format (default True).

create_new_data()
Create new feature for a given complex.

update_feature()
Update existing feature in a complex.

read_pdb()
Create a sql databse for the pdb.

get_contact_center()
Get the center of conact atoms.

add_all_features()
Add all the features toa given molecule.

add_all_atomic_densities()
Add all atomic densities.

define_grid_points()
Define the grid points.

map_atomic_densities(*only_contact=True*)
Map the atomic densities to the grid.

Parameters **only_contact** (*bool, optional*) – Map only the contact atoms

Raises ImportError – Description

densgrid(*center*, *vdw_radius*)
Function to map individual atomic density on the grid.

The formula is equation (1) of the Koes paper Protein-Ligand Scoring with Convolutional NN Arxiv:1612.02751v1

Parameters

- **center** (*list(float)*) – position of the atoms
- **vdw_radius** (*float*) – vdw radius of the atom

Returns np.array (mapped density)

Return type TYPE

map_features(*featlist*, *transform=None*)
Map individual feature to the grid.

For residue based feature the feature file must be of the format chainID residue_name(3-letter) residue_number [values]

For atom based feature it must be chainID residue_name(3-letter) residue_number atome_name [values]

Parameters

- **featlist** (*list(str)*) – list of features to be mapped
- **transform** (*callable, optional*) – transformation of the feature (?)

Returns Mapped features

Return type np.array

Raises

- ImportError – Description
- ValueError – Description

featgrid(*center*, *value*, *type_='fast_gaussian'*)
Map an individual feature (atomic or residue) on the grid.

Parameters

- **center** (*list(float)*) – position of the feature center
- **value** (*float*) – value of the feature
- **type** (*str, optional*) – method to map

Returns Mapped feature

Return type np.array

Raises ValueError – Description

export_grid_points()
export the grid points to the hdf5 file.

hdf5_grid_data(*dict_data*, *data_name*)
Save the mapped feature to the hdf5 file.

Parameters

- **dict_data** (*dict*) – feature values stored as a dict
- **data_name** (*str*) – feature name

2.1.2 Learning

This module contains all the tools for deep learning in DeepRank. The two main modules are `deeprank.learn.DataSet` and `deeprank.learn.NeuralNet`. The `DataSet` class allows to process several hdf5 files created by the `deeprank.generate` toolset for use by pyTorch. This is done by creating several `torch.data_utils.DataLoader` for the training, valiation and test of the model. Several options are possible to specify and filter which conformations should be used in the dataset. The `NeuralNet` class is in charge of the deep learning part.There as well several options are possible to specify the task to be performed, the architecture of the neural network etc ....

Example:

```
>>> from deeprank.learn import *
>>> from model3d import cnn_class
>>>
>>> database = '1ak4.hdf5'
>>>
>>> # declare the dataset instance
>>> data_set = DataSet(database,
>>>                    chain1='C',
>>>                    chain2='D',
>>>                    select_feature='all',
>>>                    select_target='IRMSD',
>>>                    dict_filter={'IRMSD':'<4. or >10.'})
>>>
>>>
>>> # create the network
>>> model = NeuralNet(data_set, cnn_class, model_type='3d', task='class')
>>>
>>> # start the training
>>> model.train(nepoch = 250,divide_trainset=0.8, train_batch_size = 50, num_
↪workers=8)
>>>
>>> # save the model
>>> model.save_model()
```

The details of the submodules are presented here. The two main ones are `deeprank.learn.DataSet` and `deeprank.learn.NeuralNet`.

note The module `deeprank.learn.modelGenerator` can automatically create the file defining the neural network architecture.

DataSet: create a torch dataset

class deeprank.learn.DataSet.**DataSet**(*train_database*, *valid_database=None*, *test_database=None*, *chain1='A'*, *chain2='B'*, *mapfly=True*, *grid_info=None*, *use_rotation=None*, *select_feature='all'*, *select_target='DOCKQ'*, *normalize_features=True*, *normalize_targets=True*, *target_ordering=None*, *dict_filter=None*, *pair_chain_feature=None*, *transform_to_2D=False*, *projection=0*, *clip_features=True*, *clip_factor=1.5*, *rotation_seed=None*, *tqdm=False*, *process=True*)

Generates the dataset needed for pytorch.

This class hanldes the data generated by deeprank.generate to be used in the deep learning part of DeepRank.

Parameters

- **train_database** (*list(str)*) – names of the hdf5 files used for the training/validation. Example: ['1AK4.hdf5','1B7W.hdf5',...]
- **valid_database** (*list(str)*) – names of the hdf5 files used for the validation. Example: ['1ACB.hdf5','4JHF.hdf5',...]
- **test_database** (*list(str)*) – names of the hdf5 files used for the test. Example: ['7CEI.hdf5']
- **chain1** (*str*) – first chain ID, defaults to 'A'

- **chain2** (*str*) – second chain ID, defaults to 'B'
- **mapfly** (*bool*) – do we compute the map in the batch preparation or read them
- **grid_info** (*dict*) – grid information to map the feature. If None the original grid points are used. The dict contains:
  - 'number_of_points", the shape of grid
  - 'resolution', the resolution of grid, unit in A

  #### Example

  {'number_of_points': [10, 10, 10], 'resolution': [3, 3, 3]}
- **use_rotation** (*int*) – number of rotations to use. Example: 0 (use only original data) Default: None (use all data of the database)
- **select_feature** (*dict or 'all', optional*) – Select the features used in the learning. if mapfly is True: - {'AtomDensities': 'all', 'Features': 'all'} - {'AtomicDensities': config.atom_vdw_radius_noH, 'Features': ['PSSM_*', 'pssm_ic_*']} if mapfly is False: - {'AtomDensities_ind': 'all', 'Feature_ind': 'all'} - {'Feature_ind': ['PSSM_*', 'pssm_ic_*']} Default: 'all'
- **select_target** (*str,optional*) – Specify required target. Default: 'DOCKQ'
- **normalize_features** (*Bool, optional*) – normalize features or not Default: True
- **normalize_targets** (*Bool, optional*) – normalize targets or not Default: True
- **target_ordering** (*str*) – 'lower' (the lower the better) or 'higher' (the higher the better) By default is not specified (None) and the code tries to identify it. If identification fails 'lower' is used.
- **dict_filter** (*None or dict, optional*) – Specify if we filter the complexes based on target values, Example: {'IRMSD': '<4. or >10'} (select complexes with IRMSD lower than 4 or larger than 10) Default: None
- **pair_chain_feature** (*None or callable, optional*) – method to pair features of chainA and chainB Example: np.sum (sum the chainA and chainB features)
- **transform_to_2D** (*bool, optional*) – Boolean to use 2d maps instead of full 3d Default: False
- **projection** (*int*) – Projection axis from 3D to 2D: Mapping: 0 -> yz, 1 -> xz, 2 -> xy Default = 0
- **clip_features** (*bool, optional*) – Remove too large values of the grid. Can be needed for native complexes where the coulomb feature might be too large
- **clip_factor** (*float, optional*) – the features are clipped at: +/-mean + clip_factor * std
- **tqdm** (*bool, optional*) – Print the progress bar
- **process** (*bool, optional*) – Actually process the data set. Must be set to False when reusing a model for testing
- **rotation_seed** (*int, optional*) – random seed for getting rotation axis and angle.

Examples

```
>>> from deeprank.learn import *
>>> train_database = '1ak4.hdf5'
>>> data_set = DataSet(train_database,
>>>                    valid_database = None,
>>>                    test_database = None,
>>>                    chain1='C',
>>>                    chain2='D',
>>>                    grid_info = {
>>>                        'number_of_points': (10, 10, 10),
>>>                        'resolution': (3, 3, 3)
>>>                    },
>>>                    select_feature = {
>>>                       'AtomicDensities': 'all',
>>>                       'Features': [
>>>                            'PSSM_*', 'pssm_ic_*' ]
>>>                    },
>>>                    select_target='IRMSD',
>>>                    normalize_features = True,
>>>                    normalize_targets=True,
>>>                    pair_chain_feature=np.add,
>>>                    dict_filter={'IRMSD':'<4. or >10.'},
>>>                    process = True)
```

static _get_database_name(*database*)
Get the list of hdf5 database file names.

Parameters **database** (*None, str or list(str)*) – hdf5 database name(s).

Returns hdf5 file names

Return type list

process_dataset()
Process the data set.

Done by default. However must be turned off when one want to test a pretrained model. This can be done by setting `process=False` in the creation of the `DataSet` instance.

static check_hdf5_files(*database*)
Check if the data contained in the hdf5 file is ok.

create_index_molecules()
Create the indexing of each molecule in the dataset.

Create the indexing: [('1ak4.hdf5,1AK4_100w),...,('1fqj.hdf5,1FGJ_400w)] This allows to refer to one complex with its index in the list.

Raises `ValueError` – No aviable training data after filtering.

_select_pdb(*mol_names*)
Select complexes.

Parameters **mol_names** (*list*) – list of complex names

Returns list of selected complexes

Return type list

filter(*molgrp*)
Filter the molecule according to a dictionary, e.g., dict_filter={'DOCKQ':'>0.1', 'IRMSD':'<=4 or

>10'}).

The filter is based on the attribute self.dict_filter that must be either of the form: { 'name': cond } or None

Parameters **molgrp** (*str*) – group name of the molecule in the hdf5 file

Returns True if we keep the complex False otherwise

Return type bool

Raises ValueError – If an unsuported condition is provided

get_mapped_feature_name()

Get actual mapped feature names for feature selections.

Note:

- **class parameter self.select_feature examples:**
  - 'all'
  - {'AtomicDensities_ind': 'all', 'Feature_ind':all}
  - {'Feature_ind': ['PSSM_*', 'pssm_ic_*']}
- Feature type must be: 'AtomicDensities_ind' or 'Feature_ind'.

Raises

- KeyError – Wrong feature type.
- KeyError – Wrong feature type.

get_raw_feature_name()

Get actual raw feature names for feature selections.

Note:

- **class parameter self.select_feature examples:**
  - 'all'
  - {'AtomicDensities': 'all', 'Features':all}
  - {'AtomicDensities': config.atom_vaw_radius_noH, 'Features': ['PSSM_*', 'pssm_ic_*']}
- Feature type must be: 'AtomicDensities' or 'Features'.

Raises

- KeyError – Wrong feature type.
- KeyError – Wrong feature type.

print_possible_features()

Print the possible features in the group.

get_pairing_feature()

Creates the index of paired features.

get_input_shape()
Get the size of the data and input.

Note:

- self.data_shape: shape of the raw 3d data set
- self.input_shape: input size of the CNN. Potentially after 2d transformation.

---

get_grid_shape()
Get the shape of the matrices.

Raises `ValueError` – If no grid shape is provided or is present in the HDF5 file

compute_norm()
compute the normalization factors.

get_norm()
Get the normalization values for the features.

_read_norm()
Read or create the normalization file for the complex.

_get_target_ordering(*order*)
Determine if ordering of the target.

This can be lower the better or higher the better If it can't determine the ordering 'lower' is assumed

backtransform_target(*data*)
Returns the values of the target after de-normalization.

Parameters **data** (*list(float)*) – normalized data

Returns un-normalized data

Return type list(float)

_normalize_target(*target*)
Normalize the values of the targets.

Parameters **target** (*list(float)*) – raw data

Returns normalized data

Return type list(float)

_normalize_feature(*feature*)
Normalize the values of the features.

Parameters **feature** (*np.array*) – raw feature values

Returns normalized feature values

Return type np.array

_clip_feature(*feature*)
Clip the value of the features at +/- mean + clip_factor * std. :param feature: raw feature values :type feature: np.array

Returns clipped feature values

Return type np.array

static _mad_based_outliers(*points*, *minv*, *maxv*, *thresh=3.5*)
Mean absolute deviation based outlier detection.

(Experimental). :param points: raw input data :type points: np.array :param minv: Minimum (negative) value requested :type minv: float :param maxv: Maximum (positive) value requested :type maxv: float :param thresh: Threshold for data detection :type thresh: float, optional

Returns data where outliers were replaced by min/max values

Return type TYPE

load_one_molecule(*fname*, *mol=None*)
Load the feature/target of a single molecule.

Parameters

- **fname** (*str*) – hdf5 file name
- **mol** (*None or str, optional*) – name of the complex in the hdf5

Returns features, targets

Return type np.array,float

map_one_molecule(*fname*, *mol=None*, *angle=None*, *axis=None*)
Map the feature and load feature/target of a single molecule.

Parameters

- **fname** (*str*) – hdf5 file name
- **mol** (*None or str, optional*) – name of the complex in the hdf5

Returns features, targets

Return type np.array,float

static convert2d(*feature*, *proj2d*)
Convert the 3D volumetric feature to a 2D planar data set.

proj2d specifies the dimension that we want to consider as channel for example for proj2d = 0 the 2D images are in the yz plane and the stack along the x dimension is considered as extra channels :param feature: raw features :type feature: np.array :param proj2d: projection :type proj2d: int

Returns projected features

Return type np.array

static make_feature_pair(*feature*, *op*)
Pair the features of both chains.

Parameters

- **feature** (*np.array*) – raw features
- **op** (*callable*) – function to combine the features

Returns combined features

Return type np.array

Raises ValueError – if op is not callable

get_grid(*mol_data*)
Get meshed grids and number of pointgs

Parameters **mol_data** (*h5 group*) – HDF5 moleucle group

Raises ValueError – Grid points not found in mol_data.

Returns meshgrid, npts

Return type tuple, tuple

map_atomic_densities(*feat_names*, *mol_data*, *grid*, *npts*, *angle*, *axis*)
Map atomic densities.

Parameters

- **feat_names** (*dict*) – Element type and vdw radius
- **mol_data** (*h5 group*) – HDF5 molecule group
- **grid** (*tuple*) – mesh grid of x,y,z
- **npts** (*tuple*) – number of points on axis x,y,z
- **angle** (*float*) – rotation angle
- **axis** (*list*) – rotation axis

Returns atomic densities of each atom type on each chain

Return type list

static _densgrid(*center*, *vdw_radius*, *grid*, *npts*)
Function to map individual atomic density on the grid.

The formula is equation (1) of the Koes paper Protein-Ligand Scoring with Convolutional NN Arxiv:1612.02751v1

Parameters

- **center** (*list(float)*) – position of the atoms
- **vdw_radius** (*float*) – vdw radius of the atom

Returns np.array (mapped density)

Return type TYPE

map_feature(*feat_names*, *mol_data*, *grid*, *npts*, *angle*, *axis*)

static _featgrid(*center*, *value*, *grid*, *npts*)
Map an individual feature (atomic or residue) on the grid.

Parameters

- **center** (*list(float)*) – position of the feature center
- **value** (*float*) – value of the feature
- **type** (*str, optional*) – method to map

Returns Mapped feature

Return type np.array

Raises ValueError – Description

NeuralNet: perform deep learning

class deeprank.learn.NeuralNet.**NeuralNet**(*data_set*, *model*, *model_type='3d'*, *proj2d=0*, *task='reg'*, *class_weights=None*, *pretrained_model=None*, *chain1='A'*, *chain2='B'*, *cuda=False*, *ngpu=0*, *plot=False*, *save_hitrate=False*, *save_classmetrics=False*, *outdir='./'*)

Train a Convolutional Neural Network for DeepRank.

Parameters

- **data_set** (*deeprank.DataSet or list(str)*) – Data set used for training or testing. - deeprank.DataSet for training; - str or list(str), e.g. 'x.hdf5', ['x1.hdf5', 'x2.hdf5'], for testing when pretrained model is loaded.
- **model** (*nn.Module*) – Definition of the NN to use. Must subclass nn.Module. See examples in model2d.py and model3d.py
- **model_type** (*str*) – Type of model we want to use. Must be '2d' or '3d'. If we specify a 2d model, the data set is automatically converted to the correct format.
- **proj2d** (*int*) – Defines how to slice the 3D volumetric data to generate 2D data. Allowed values are 0, 1 and 2, which are to slice along the YZ, XZ or XY plane, respectively.
- **task** (*str 'reg' or 'class'*) – Task to perform. - 'reg' for regression - 'class' for classification. The loss function, the target datatype and plot functions will be autmatically adjusted depending on the task.
- **class_weights** (*Tensor*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size #classes. Only applicable on 'class' task.
- **pretrained_model** (*str*) – Saved model to be used for further training or testing. When using pretrained model, remember to set the following 'chain1' and 'chain2' for the new data.
- **chain1** (*str*) – first chain ID of new data when using pretrained model
- **chain2** (*str*) – second chain ID of new data when using pretrained model
- **cuda** (*bool*) – Use CUDA.
- **ngpu** (*int*) – number of GPU to be used.
- **plot** (*bool*) – Plot the prediction results.
- **save_hitrate** (*bool*) – Save and plot hit rate.
- **save_classmetrics** (*bool*) – Save and plot classification metrics. Classification metrics include: - accuracy(ACC) - sensitivity(TPR) - specificity(TNR)
- **outdir** (*str*) – output directory

Raises

- ValueError – if dataset format is not recognized
- ValueError – if task is not recognized

Examples

Train models: >>> data_set = Dataset(...) >>> model = NeuralNet(data_set, cnn, ... model_type='3d', task='reg', ... plot=True, save_hitrate=True, ... outdir='./out/') >>> model.train(nepoch = 50, divide_trainset=0.8, ... train_batch_size = 5, num_workers=0)

Test a model on new data: >>> data_set = ['test01.hdf5', 'test02.hdf5'] >>> model = NeuralNet(data_set, cnn, ... pretrained_model = './model.pth.tar', ... outdir='./out/') >>> model.test()

train(*nepoch=50*, *divide_trainset=None*, *hdf5='epoch_data.hdf5'*, *train_batch_size=10*, *preshuffle=True*, *preshuffle_seed=None*, *export_intermediate=True*, *num_workers=1*, *save_model='best'*, *save_epoch='intermediate'*, *hit_cutoff=None*)

Perform a simple training of the model.

Parameters

- **nepoch** (*int, optional*) – number of iterations
- **divide_trainset** (*list, optional*) – the percentage assign to the training, validation and test set. Examples: [0.7, 0.2, 0.1], [0.8, 0.2], None
- **hdf5** (*str, optional*) – file to store the training results
- **train_batch_size** (*int, optional*) – size of the batch
- **preshuffle** (*bool, optional*) – preshuffle the dataset before dividing it.
- **preshuffle_seed** (*int, optional*) – set random seed for preshuffle
- **export_intermediate** (*bool, optional*) – export data at intermediate epochs.
- **num_workers** (*int, optional*) – number of workers to be used to prepare the batch data
- **save_model** (*str, optional*) – 'best' or 'all', save only the best model or all models.
- **save_epoch** (*str, optional*) – 'intermediate' or 'all', save the epochs data to HDF5.
- **hit_cutoff** (*float, optional*) – the cutoff used to define hit by comparing with docking models' target value, e.g. IRMSD value

static convertSeconds2Days(*time*)

test(*hdf5='test_data.hdf5'*, *hit_cutoff=None*, *has_target=False*)

Test a predefined model on a new dataset.

Parameters

- **hdf5** (*str, optional*) – hdf5 file to store the test results
- **hit_cutoff** (*float, optional*) – the cutoff used to define hit by comparing with docking models' target value, e.g. IRMSD value
- **has_target** (*bool, optional*) – specify the presence (True) or absence (False) of target values in the test set. No metrics can be computed if False.

Examples

```
>>> # adress of the database
>>> database = '1ak4.hdf5'
>>> # Load the model in a new network instance
>>> model = NeuralNet(database, cnn,
...                   pretrained_model='./model/model.pth.tar',
...                   outdir='./test/')
>>> # test the model
>>> model.test()
```

save_model (*filename='model.pth.tar'*)
save the model to disk.

Parameters **filename** (*str, optional*) – name of the file

load_model_params ()
Get model parameters from a saved model.

load_optimizer_params ()
Get optimizer parameters from a saved model.

load_nn_params ()
Get NeuralNet parameters from a saved model.

load_data_params ()
Get dataset parameters from a saved model.

_divide_dataset (*divide_set*, *preshuffle*, *preshuffle_seed*)
Divide the data set into training, validation and test according to the percentage in divide_set.

Parameters

- **divide_set** (*list(float)*) – percentage used for training/validation/test. Example: [0.8, 0.1, 0.1], [0.8, 0.2]
- **preshuffle** (*bool*) – shuffle the dataset before dividing it
- **preshuffle_seed** (*int, optional*) – set random seed

Returns

Indices of the training/validation/test set.

Return type list(int),list(int),list(int)

_train (*index_train*, *index_valid*, *index_test*, *nepoch=50*, *train_batch_size=5*, *export_intermediate=False*, *num_workers=1*, *save_epoch='intermediate'*, *save_model='best'*)
Train the model.

Parameters

- **index_train** (*list(int)*) – Indices of the training set
- **index_valid** (*list(int)*) – Indices of the validation set
- **index_test** (*list(int)*) – Indices of the testing set
- **nepoch** (*int, optional*) – numbr of epoch
- **train_batch_size** (*int, optional*) – size of the batch
- **export_intermediate** (*bool, optional*) – export itnermediate data
- **num_workers** (*int, optional*) – number of workers pytorch uses to create the batch size
- **save_epoch** (*str,optional*) – 'intermediate' or 'all'

- **save_model** (*str, optional*) – 'all' or 'best'

Returns Parameters of the network after training

Return type torch.tensor

_epoch (*data_loader*, *train_model*, *has_target=True*)
Perform one single epoch iteration over a data loader.

Parameters

- **data_loader** (*torch.DataLoader*) – DataLoader for the epoch
- **train_model** (*bool*) – train the model if True or not if False

Returns loss of the model dict: data of the epoch

Return type float

_get_variables (*inputs*, *targets*)
Convert the feature/target in torch.Variables.

The format is different for regression where the targets are float and classification where they are int.

Parameters

- **inputs** (*np.array*) – raw features
- **targets** (*np.array*) – raw target values

Returns features torch.Variable: target values

Return type torch.Variable

_export_losses (*figname*)
Plot the losses vs the epoch.

Parameters **figname** (*str*) – name of the file where to export the figure

_export_metrics (*metricname*)

_plot_scatter_reg (*figname*)
Plot a scatter plots of predictions VS targets.

Useful to visualize the performance of the training algorithm

Parameters **figname** (*str*) – filename

_plot_boxplot_class (*figname*)
Plot a boxplot of predictions VS targets.

It is only usefull in classification tasks.

Parameters **figname** (*str*) – filename

plot_hit_rate (*figname*)
Plot the hit rate of the different training/valid/test sets.

The hit rate is defined as: The percentage of positive(near-native) decoys that are included among the top m decoys.

Parameters **figname** (*str*) – filename for the plot

_compute_hitrate (*hit_cutoff=None*)

_get_relevance (*data*, *hit_cutoff=None*)

_get_classmetrics (*data*, *metricname*)

static _get_binclass_prediction(*data*)

_export_epoch_hdf5(*epoch*, *data*)

Export the epoch data to the hdf5 file.

Export the data of a given epoch in train/valid/test group. In each group are stored the predcited values (outputs), ground truth (targets) and molecule name (mol).

Parameters

- **epoch** (*int*) – index of the epoch
- **data** (*dict*) – data of the epoch

modelGenerator: generate NN architecture

class deeprank.learn.modelGenerator.**NetworkGenerator**(*name='_tmp_model_'*, *fname='_tmp_model_.py'*, *conv_layers=None*, *fc_layers=None*)

Automatic generation of NN files.

This class allows for automatic generation of python file containing the definition of torch formatted neural network.

Parameters

- **name** (*str, optional*) – name of the model in the python file
- **fname** (*str, optional*) – name of the file containing the model
- **conv_layers** (*list(layers)*) – list of convolutional layers
- **fc_layers** (*list(layers)*) – list of fully connected layers

Example

```
>>> conv_layers = []
>>> conv_layers.append(conv(output_size=4,kernel_size=2,post='relu'))
>>> conv_layers.append(pool(kernel_size=2))
>>> conv_layers.append(conv(input_size=4,output_size=5,kernel_size=2,post='relu'))
>>> conv_layers.append(pool(kernel_size=2))
>>>
>>> fc_layers = []
>>> fc_layers.append(fc(output_size=84,post='relu'))
>>> fc_layers.append(fc(input_size=84,output_size=1))
>>>
>>> MG = NetworkGenerator(name='test',fname='model_test.py',conv_layers=conv_
↪layers,fc_layers=fc_layers)
>>> MG.print()
>>> MG.write()
```

write()

Write the model to file.

static _write_import(*fhandle*)

_write_definition(*fhandle*)

_write_init(*fhandle*)

static _write_conv_output(*fhandle*)

_write_forward_feature(*fhandle*)

_write_forward(*fhandle*)

print()
Print the model to screen.

get_new_random_model()
Get a new Random Model.

_init_conv_layer_random(*ilayer*)

_init_fc_layer_random(*ilayer*)

class deeprank.learn.modelGenerator.**conv**(*input_size=-1*, *output_size=None*, *kernel_size=None*, *post=None*)
Wrapper around the convolutional layer.

Parameters

- **input_size** (*int, optional*) – input size (default, let the generator figure it out)
- **output_size** (*int, optional*) – output size
- **kernel_size** (*int, optional*) – kernel size
- **post** (*str, optional*) – post process of the data

Example:

```
>>> conv_layers.append(conv(output_size=4,kernel_size=2,post='relu'))
```

class deeprank.learn.modelGenerator.**pool**(*kernel_size=None*, *post=None*)
Wrapper around the pool layer.

Parameters

- **kernel_size** (*int, optional*) – kernel size
- **post** (*str, optional*) – post process of the data

Example:

```
>>> conv_layers.append(pool(kernel_size=2))
```

class deeprank.learn.modelGenerator.**dropout**(*percent=0.5*)
Wrapper around the dropout layer layer.

Parameters **percent** (*float*) – percent of dropout

Example:

```
>>> fc_layers.append(dropout(precent=0.25))
```

class deeprank.learn.modelGenerator.**fc**(*input_size=-1*, *output_size=None*, *post=None*)
Wrapper around the fully conneceted layer.

Parameters

- **input_size** (*int, optional*) – input size (default, let the generator figure it out)
- **output_size** (*int, optional*) – output size
- **post** (*str, optional*) – post process of the data

Example:

```
>>> fc_layers.append(fc(output_size=84,post='relu'))
```

2.1.3 Features

This module contains all the tools to compute feature values for molecular structure. Each submodule must be subclass `deepra`

- `AtomicFeatures`:Coulomb, van der Waals interactions and atomic charges
- `BSA` : Burried Surface area
- `FullPSSM` : Complete PSSM data
- `PSSM_IC` : Information content of the PSSM
- `ResidueDensity` : The residue density for polar/apolar/charged pairs

As you can see in the source each python file contained a `__compute_feature__` function. This is the function called in `deeprank.generate`.

Here are detailed the class in charge of feature calculations.

Atomic Feature

class deeprank.features.AtomicFeature.**AtomicFeature**(*pdbfile*, *chain1='A'*, *chain2='B'*, *param_charge=None*, *param_vdw=None*, *patch_file=None*, *contact_cutoff=8.5*, *verbose=False*)

Compute the Coulomb, van der Waals interaction and charges.

Parameters

- **pdbfile** (*str*) – pdb file of the molecule
- **chain1** (*str*) – First chain ID, defaults to 'A'
- **chain2** (*str*) – Second chain ID, defaults to 'B'
- **param_charge** (*str*) – file name of the force field file containing the charges e.g. protein-allhdg5.4_new.top. Must be of the format: * CYM atom O type=O charge=-0.500 end * ALA atom N type=NH1 charge=-0.570 end
- **param_vdw** (*str*) – file name of the force field containing vdw parameters e.g. protein-allhdg5.4_new.param. Must be of the format: * NONBonded CYAA 0.105 3.750 0.013 3.750 * NONBonded CCIS 0.105 3.750 0.013 3.750
- **patch_file** (*str*) – file name of a valid patch file for the parameters e.g. patch.top. The way we handle the patching is very manual and should be made more automatic.
- **contact_cutoff** (*float*) – the maximum distance in Å between 2 contact atoms.
- **verbose** (*bool*) – print or not.

Examples

```
>>> pdb = '1AK4_100w.pdb'
>>>
>>> # get the force field included in deeprank
>>> # if another FF has been used to compute the ref
>>> # change also this path to the correct one
>>> FF = pkg_resources.resource_filename(
>>>     'deeprank.features','') + '/forcefield/'
>>>
>>> # declare the feature calculator instance
>>> atfeat = AtomicFeature(pdb,
>>>    param_charge = FF + 'protein-allhdg5-4_new.top',
>>>    param_vdw    = FF + 'protein-allhdg5-4_new.param',
>>>    patch_file   = FF + 'patch.top')
>>>
>>> # assign parameters
>>> atfeat.assign_parameters()
>>>
>>> # only compute the pair interactions here
>>> atfeat.evaluate_pair_interaction(save_interactions=test_name)
>>>
>>> # close the db
>>> atfeat.sqldb._close()
```

read_charge_file()
Read the .top file given in entry.

This function creates:

- self.charge: dictionary {(resname,atname):charge}
- self.valid_resnames: list ['VAL','ALP', ...]
- self.at_name_type_convertor: dict {(resname,atname):attype}

read_patch()
Read the patchfile.

This function creates

- self.patch_charge: Dict {(resName,atName): charge}
- self.patch_type : Dict {(resName,atName): type}

read_vdw_file()
Read the .param file.

The param file must be of the form:

NONBONDED ATNAME 0.10000 3.298765 0.100000 3.089222

- First two numbers are for inter-chain interations
- Last two nmbers are for intra-chain interactions (We only compute the interchain here)

This function creates

- self.vdw: dictionary {attype:[E1,S1]}

get_contact_atoms()
Get the contact atoms only select amino acids.

The ligands are not considered.

_extend_contact_to_residue()
Extend the contact atoms to entire residue where one atom is contacting.

assign_parameters()
Assign to each atom in the pdb its charge and vdw interchain parameters.

Directly deals with the patch so that we don't loop over the residues multiple times.

static _get_altResName(*resName*, *atNames*)
Apply the patch data.

This is adopted from preScan.pl This is very static and I don't quite like it The structure of the dictionary is as following

{ **NEWRESTYPE: 'OLDRESTYPE',** [atom types that must be present], [atom types that must NOT be present]]}

Parameters

- **resName** (*str*) – name of the residue
- **atNames** (*list (str)*) – names of the atoms

_get_vdw(*resName*, *altResName*, *atNames*)
Get vdw itneraction terms.

Parameters

- **resName** (*str*) – name of the residue
- **altResName** (*str*) – alternative name of the residue
- **atNames** (*list (str)*) – names of the atoms

_get_charge(*resName*, *altResName*, *atNames*)
Get the charge information.

Parameters

- **resName** (*str*) – name of the residue
- **altResName** (*str*) – alternative name of the residue
- **atNames** (*list (str)*) – names of the atoms

evaluate_charges(*extend_contact_to_residue=False*)
Evaluate the charges.

Parameters **extend_contact_to_residue** (*bool, optional*) – extend to res

evaluate_pair_interaction(*print_interactions=False*, *save_interactions=False*)
Evalaute the pair interactions (coulomb and vdw).

Parameters

- **print_interactions** (*bool, optional*) – print data to screen
- **save_interactions** (*bool, optional*) – save the itneractions to file.

compute_coulomb_interchain_only(*dosum=True*, *contact_only=False*)
Compute the coulomb interactions between the chains only.

Parameters

- **dosum** (*bool, optional*) – sum the interaction terms for each atoms
- **contact_only** (*bool, optional*) – consider only contact atoms

compute_vdw_interchain_only(*dosum=True*, *contact_only=False*)
Compute the vdw interactions between the chains only.

Parameters

- **dosum** (*bool, optional*) – sum the interaction terms for each atoms
- **contact_only** (*bool, optional*) – consider only contact atoms

static _prefactor_vdw(*r*)
prefactor for vdw interactions.

deeprank.features.AtomicFeature.**__compute_feature__**(*pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*)
Main function called in deeprank for the feature calculations.

Parameters

- **pdb_data** (*list(bytes)*) – pdb information
- **featgrp** (*str*) – name of the group where to save xyz-val data
- **featgrp_raw** (*str*) – name of the group where to save human readable data
- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID

Buried Surface Area

class deeprank.features.BSA.**BSA**(*pdb_data*, *chain1='A'*, *chain2='B'*)
Compute the burried surface area feature.

Freesasa is required for this feature. From Freesasa version 2.0.3 the Python bindings are released as a separate module. They can be installed using >>> pip install freesasa

Parameters

- **pdb_data** (*list(byte) or str*) – pdb data or pdb filename
- **chain1** (*str, optional*) – name of the first chain
- **chain2** (*str, optional*) – name of the second chain

Example

```
>>> bsa = BSA('1AK4.pdb')
>>> bsa.get_structure()
>>> bsa.get_contact_residue_sasa()
>>> bsa.sql._close()
```

get_structure()
Get the pdb structure of the molecule.

get_contact_residue_sasa(*cutoff=5.5*)
Compute the feature of BSA.

It generates following feature: bsa

Raises ValueError – No interface residues found.

deeprank.features.BSA.**__compute_feature__**(*pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*)

Main function called in deeprank for the feature calculations.

Parameters

- **pdb_data** (*list(bytes)*) – pdb information
- **featgrp** (*str*) – name of the group where to save xyz-val data
- **featgrp_raw** (*str*) – name of the group where to save human readable data
- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID

FullPSSM

class deeprank.features.FullPSSM.**FullPSSM**(*mol_name=None*, *pdb_file=None*, *chain1='A'*, *chain2='B'*, *pssm_path=None*, *pssm_format='new'*, *out_type='pssmvalue'*)

Compute all the PSSM data.

Simply extracts all the PSSM information and store that into features

Parameters

- **mol_name** (*str*) – name of the molecule. Defaults to None.
- **pdb_file** (*str*) – name of the pdb_file. Defaults to None.
- **chain1** (*str*) – First chain ID. Defaults to 'A'
- **chain2** (*str*) – Second chain ID. Defaults to 'B'
- **pssm_path** (*str*) – path to the pssm data. Defaults to None.
- **pssm_format** (*str*) – "old" or "new" pssm format. Defaults to 'new'.
- **out_type** (*str*) – which feature to generate, 'pssmvalue' or 'pssmic'. Defaults to 'pssmvalue'. 'pssm_format' must be 'new' when set type is 'pssmic'.

Examples

```
>>> path = '/home/test/PSSM_newformat/'
>>> pssm = FullPSSM(mol_name='2ABZ',
>>>                 pdb_file='2ABZ_1w.pdb',
>>>                 pssm_path=path)
>>> pssm.read_PSSM_data()
>>> pssm.get_feature_value()
>>> print(pssm.feature_data_xyz)
```

static get_ref_mol_name(*mol_name*)

Get the bared mol name.

read_PSSM_data()

Read the PSSM data into a dictionary.

get_feature_value(*cutoff=5.5*)

get the feature value.

deeprank.features.FullPSSM.**__compute_feature__**(*pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*, *out_type='pssmvalue'*)

Main function called in deeprank for the feature calculations.

Parameters

- **pdb_data** (*list(bytes)*) – pdb information
- **featgrp** (*str*) – name of the group where to save xyz-val data
- **featgrp_raw** (*str*) – name of the group where to save human readable data
- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID
- **out_type** (*str*) – which feature to generate, 'pssmvalue' or 'pssmic'.

PSSM Information Content

class deeprank.features.PSSM_IC.**PSSM_IC**(*mol_name=None*, *pdb_file=None*, *chain1='A'*, *chain2='B'*, *pssm_path=None*, *pssm_format='new'*, *out_type='pssmvalue'*)

Compute all the PSSM data.

Simply extracts all the PSSM information and store that into features

Parameters

- **mol_name** (*str*) – name of the molecule. Defaults to None.
- **pdb_file** (*str*) – name of the pdb_file. Defaults to None.
- **chain1** (*str*) – First chain ID. Defaults to 'A'
- **chain2** (*str*) – Second chain ID. Defaults to 'B'
- **pssm_path** (*str*) – path to the pssm data. Defaults to None.
- **pssm_format** (*str*) – "old" or "new" pssm format. Defaults to 'new'.
- **out_type** (*str*) – which feature to generate, 'pssmvalue' or 'pssmic'. Defaults to 'pssmvalue'. 'pssm_format' must be 'new' when set type is 'pssmic'.

Examples

```
>>> path = '/home/test/PSSM_newformat/'
>>> pssm = FullPSSM(mol_name='2ABZ',
>>>                 pdb_file='2ABZ_1w.pdb',
>>>                 pssm_path=path)
>>> pssm.read_PSSM_data()
>>> pssm.get_feature_value()
>>> print(pssm.feature_data_xyz)
```

deeprank.features.PSSM_IC.**__compute_feature__**(*pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*)

Contact Residue Density

class deeprank.features.ResidueDensity.**ResidueDensity**(*pdb_data*, *chain1='A'*, *chain2='B'*)

Compute the residue contacts between polar/apolar/charged residues.

Parameters

- **pdb_data** (*list(byte) or str*) – pdb data or pdb filename
- **chain1** (*str*) – First chain ID. Defaults to 'A'
- **chain2** (*str*) – Second chain ID. Defaults to 'B'

Example

```
>>> rcd = ResidueDensity('1EWY_100w.pdb')
>>> rcd.get(cutoff=5.5)
>>> rcd.extract_features()
```

get(*cutoff=5.5*)

Get residue contacts.

Raises ValueError – No residue contact found.

extract_features()

Compute the feature of residue contacts between polar/apolar/charged residues.

It generates following features: RCD_apolar-apolar RCD_apolar-charged RCD_charged-charged RCD_polar-apolar RCD_polar-charged RCD_polar-polar RCD_total

class deeprank.features.ResidueDensity.**residue_pair**(*res*, *rtype*)

Ancillary class that holds information for a given residue.

deeprank.features.ResidueDensity.**__compute_feature__**(*pdb_data*, *featgrp*, *featgrp_raw*, *chain1*, *chain2*)

Main function called in deeprank for the feature calculations.

Parameters

- **pdb_data** (*list(bytes)*) – pdb information
- **featgrp** (*str*) – name of the group where to save xyz-val data
- **featgrp_raw** (*str*) – name of the group where to save human readable data
- **chain1** (*str*) – First chain ID
- **chain2** (*str*) – Second chain ID

Generic Feature Class

class deeprank.features.FeatureClass.**FeatureClass**(*feature_type*)

Master class from which all the other feature classes should be derived.

Arguments feature_type(str): 'Atomic' or 'Residue'

Note: Each subclass must compute:

- **self.feature_data: dictionary of features in human readable format, e.g.**

  - **for atomic features:**
    * {'coulomb': data_dict_clb, 'vdwaals': data_dict_vdw}
    * data_dict_clb = {atom_info: [values]}
    * atom_info = (chainID, resSeq, resName, name)
  - **for residue features:**
    * {'PSSM_ALA': data_dict_pssmALA, ... }
    * data_dict_pssmALA = {residue_info: [values]}
    * residue_info = (chainID, resSeq, resName, name)
- **self.feature_data_xyz: dictionary of features in xyz-val format, e.g.**
  - {'coulomb': data_dict_clb, 'vdwaals': data_dict_vdw}
  - data_dict_clb = {xyz_info: [values]}
  - xyz_info = (chainNum, x, y, z)

---

export_data_hdf5 (*featgrp*)
Export the data in xyz-val format in an HDF5 file group.

Parameters **{[hdf5_group]} -- The hdf5 group of the feature** (*featgrp*) –

Note:

- **For atomic features, the format of the data must be:** {(chainID, resSeq, resName, name): [values]}
- **For residue features, the format must be:** {(chainID, resSeq, resName): [values]}

---

export_dataxyz_hdf5 (*featgrp*)
Export the data in xyz-val format in an HDF5 file group.

Parameters **{[hdf5_group]} -- The hdf5 group of the feature** (*featgrp*) –

static get_residue_center (*sql, centers=['CB', 'CA', 'mean'], res=None*)
Computes the center of each residue by trying different options

Parameters **{pdb2sql} -- The pdb2sql instance** (*sql*) –

Keyword Arguments

- **{list} -- list of strings (default** (*centers*) – {['CB','CA','mean']})
- **{list} -- list of residue to be considered** (*res*) –

Raises ValueError – [description]

Returns [type] – list(res), list(xyz)

2.1.4 Targets

This module contains all the functions to compute target values for molecular structures. The implemented targets at the mome

- `binary_class`: Binary class ID
- `capri_class`: CAPRI class
- `dockQ`: DockQ
- `rmsd_fnat`: CAPRI metric IRMSD, LRMSD or FNAT

As you can see in the source each python file contained a `__compute_target__` function. This is the function called in `deeprank.generate`.

Here are detailed the function in charge of target calculations.

Binary class

`deeprank.targets.binary_class.__compute_target__`(*decoy*, *targrp*)
Calculate binary class ID using IRMSD.

Parameters

- **decoy** (*bytes*) – pdb data of the decoy
- **targrp** (*h5 file hadnle*) – HDF5 'targets' group

Examples

```
>>> f = h5py.File('1LFD.hdf5')
>>> decoy = f['1LFD_9w/complex'][()]
>>> targrp = f['1LFD_9w/targets']
```

CAPRI class

`deeprank.targets.capri_class.__compute_target__`(*decoy*, *targrp*)
Calculate CAPRI class.

CAPRI class name and ID:

- 'high': 0
- 'medium': 1
- 'accepetable': 2
- 'incorrect': 3

Parameters

- **decoy** (*bytes*) – pdb data of the decoy
- **targrp** (*hdf5 file handle*) – HDF5 'targets' group

DockQ

`deeprank.targets.dockQ.__compute_target__`(*decoy*, *targrp*)
Calculate DOCKQ.

Parameters

- **decoy** (*bytes*) – pdb data of the decoy

- **targrp** (*hdf5 file handle*) – HDF5 'targets' group

RMSDs & FNAT

deeprank.targets.rmsd_fnat.**__compute_target__**(*decoy*, *targrp*, *tarname*, *save_file=False*)
Calcuate CAPRI metric IRMSD, LRMSD or FNAT.

Parameters

- **decoy** (*bytes*) – pdb data of the decoy
- **targrp** (*hdf5 file handle*) – HDF5 'targets' group
- **tarnames** (*str*) – it must be IRMSD, LRMSD or FNAT.
- **save_file** (*bool*) – save .izone, .lzone or .ref_pairs file or not, defaults to False.

Returns value of IRMSD, LRMSD or FNAT.

Return type float

Raises

- ValueError – Wrong target name
- ValueError – native complex not exist
- ValueError – native complex has empty dataset

Examples

```
>>> f = h5py.File('1LFD.hdf5')
>>> decoy = f['1LFD_9w/complex'][()]
>>> targrp = f['1LFD_9w/targets']
```

2.1.5 Tools

This module contains a series of core tools used for the feature calculations in DeepRank. These tools are at the moment:

- sasa: a simple solvent surface area calculator
- sparse: a 3D sparse matrix engine

Here are the details of the submodule given in tools.

Solvent Accessible Surface Area

class deeprank.tools.sasa.**SASA**(*pdbfile*)
Simple class that computes Surface Accessible Solvent Area.

The method follows some of the approaches presented in:

Solvent accessible surface area approximations for rapid and accurate protein structure prediction https://link.springer.com/article/10.1007%2Fs00894-009-0454-9

Example

```
>>> sasa = SASA('1AK4_1w.pdb')
>>> NV = sasa.neighbor_vector()
>>> print(NV)
```

Parameters **pdbfile** (*str*) – PDB file of the conformation

get_center (*chain1='A'*, *chain2='B'*, *center='cb'*)
Get the center of the resiudes.

Parameters

- **chain1** (*str, optional*) – Name of the first chain
- **chain2** (*str, optional*) – Name of the second chain
- **center** (*str, optional*) – Specify the center. 'cb': the center locates on carbon beta of each residue 'center': average position of all atoms of the residue

Raises ValueError – If the center is not recpgnized

get_residue_center (*chain1='A'*, *chain2='B'*)
Compute the average position of all the residues.

Parameters

- **chain1** (*str, optional*) – Name of the first chain
- **chain2** (*str, optional*) – Name of the second chain

get_residue_carbon_beta (*chain1='A'*, *chain2='B'*)
Extract the position of the carbon beta of each residue.

Parameters

- **chain1** (*str, optional*) – Name of the first chain
- **chain2** (*str, optional*) – Name of the second chain

neighbor_vector (*lbound=3.3*, *ubound=11.1*, *chain1='A'*, *chain2='B'*, *center='cb'*)
Compute teh SASA folowing the neighbour vector approach.

The method is based on Eq on page 1097 of https://link.springer.com/article/10.1007%2Fs00894-009-0454-9

Parameters

- **lbound** (*float, optional*) – lower boubd
- **ubound** (*float, optional*) – upper bound
- **chain1** (*str, optional*) – name of the first chain
- **chain2** (*str, optional*) – name of the second chain
- **center** (*str, optional*) – specify the center (see get_residue_center)

Returns neighbouring vectors

Return type dict

neighbor_count (*lbound=4.0*, *ubound=11.4*, *chain1='A'*, *chain2='B'*, *center='cb'*)
Compute the neighbourhood count of each residue.

The method is based on Eq on page 1097 of https://link.springer.com/article/10.1007%2Fs00894-009-0454-9

Parameters

- **lbound** (*float, optional*) – lower boubd
- **ubound** (*float, optional*) – upper bound
- **chain1** (*str, optional*) – name of the first chain
- **chain2** (*str, optional*) – name of the second chain
- **center** (*str, optional*) – specify the center
- **get_residue_center)** (*(see*) –

Returns Neighborhood count

Return type dict

static neighbor_weight(*dist*, *lbound*, *ubound*)
Neighboor weight.

Parameters

- **dist** (*np.array*) – distance from neighboors
- **lbound** (*float*) – lower bound
- **ubound** (*float*) – upper bound

Returns distance

Return type float

Sparse 3D Matrix

deeprank.tools.sparse.**_printif**(*string*, *cond*)

class deeprank.tools.sparse.**FLANgrid**(*sparse=None*, *index=None*, *value=None*, *shape=None*)
Flat Array sparse matrix.

Parameters

- **sparse** (*bool, optional*) – Sparse or Not
- **index** (*list(int), optional*) – single index of each non-zero element
- **value** (*list(float), optional*) – values of non-zero elements
- **shape** (*3x3 array, optional*) – Shape of the matrix

from_dense(*data*, *beta=None*, *debug=False*)
Create a sparse matrix from a dense one.

Parameters

- **data** (*np.array*) – Dense matrix
- **beta** (*float, optional*) – threshold to determine if a sparse rep is valuable
- **debug** (*bool, optional*) – print debug information

to_dense(*shape=None*)
Create a dense matrix.

Parameters **shape** (*3x3 array, optional*) – Shape of the matrix

Returns Dense 3D matrix

Return type np.array

Raises ValueError – shape not defined

_get_single_index(*index*)

Get the single index for a single element.

get the index can be used with a map # self.index = np.array(list(map(self._get_single_index,index))).astype(index_type) # however that is remarkably slow compared to the array version

Parameters **index** (*array*) – COO index

Returns index

Return type int

_get_single_index_array(*index*)

Get the single index for multiple elements.

get the index can be used with a map # self.index = np.array(list(map(self._get_single_index,index))).astype(index_type) # however that is remarkably slow compared to the array version

Parameters **index** (*array*) – COO index

Returns index

Return type list(int)

CHAPTER 3

Indices

- genindex
- modindex

Python Module Index

d

deeprank.features.AtomicFeature, 40
deeprank.features.BSA, 43
deeprank.features.FeatureClass, 46
deeprank.features.FullPSSM, 44
deeprank.features.PSSM_IC, 45
deeprank.features.ResidueDensity, 46
deeprank.generate, 13
deeprank.generate.DataGenerator, 15
deeprank.generate.GridTools, 24
deeprank.generate.NormalizeData, 23
deeprank.learn, 26
deeprank.learn.DataSet, 27
deeprank.learn.modelGenerator, 38
deeprank.learn.NeuralNet, 34
deeprank.targets, 47
deeprank.tools, 49
deeprank.tools.sasa, 49
deeprank.tools.sparse, 51

Index

Symbols

__compute_feature__() (*in module deeprank.features.AtomicFeature*), 43
__compute_feature__() (*in module deeprank.features.BSA*), 43
__compute_feature__() (*in module deeprank.features.FullPSSM*), 44
__compute_feature__() (*in module deeprank.features.PSSM_IC*), 45
__compute_feature__() (*in module deeprank.features.ResidueDensity*), 46
__compute_target__() (*in module deeprank.targets.binary_class*), 48
__compute_target__() (*in module deeprank.targets.capri_class*), 48
__compute_target__() (*in module deeprank.targets.dockQ*), 48
__compute_target__() (*in module deeprank.targets.rmsd_fnat*), 49
_add_aug_pdb() (*deeprank.generate.DataGenerator.DataGenerator method*), 22
_add_pdb() (*deeprank.generate.DataGenerator.DataGenerator method*), 22
_clip_feature() (*deeprank.learn.DataSet.DataSet method*), 31
_compile_cuda_kernel() (*deeprank.generate.DataGenerator.DataGenerator static method*), 20
_compute_features() (*deeprank.generate.DataGenerator.DataGenerator static method*), 21
_compute_hitrate() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_compute_targets() (*deeprank.generate.DataGenerator.DataGenerator static method*), 21
_densgrid() (*deeprank.learn.DataSet.DataSet static method*), 33
_divide_dataset() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
_epoch() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_export_data() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
_export_epoch_hdf5() (*deeprank.learn.NeuralNet.NeuralNet method*), 38
_export_losses() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_export_metrics() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_extend_contact_to_residue() (*deeprank.features.AtomicFeature.AtomicFeature method*), 41
_extract_data() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
_extract_shape() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
_featgrid() (*deeprank.learn.DataSet.DataSet static method*), 33
_filter_cplx() (*deeprank.generate.DataGenerator.DataGenerator method*), 21
_get_aligned_rotation_axis_angle() (*deeprank.generate.DataGenerator.DataGenerator static method*), 22
_get_aligned_sqldb() (*deeprank.generate.DataGenerator.DataGenerator method*), 22
_get_altResName() (*deeprank.features.AtomicFeature.AtomicFeature*

static method), 42
_get_binclass_prediction() (*deeprank.learn.NeuralNet.NeuralNet static method*), 37
_get_charge() (*deeprank.features.AtomicFeature.AtomicFeature method*), 42
_get_classmetrics() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_get_cuda_function() (*deeprank.generate.DataGenerator.DataGenerator static method*), 21
_get_database_name() (*deeprank.learn.DataSet.DataSet static method*), 29
_get_grid_center() (*deeprank.generate.DataGenerator.DataGenerator method*), 19
_get_relevance() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_get_single_index() (*deeprank.tools.sparse.FLANgrid method*), 52
_get_single_index_array() (*deeprank.tools.sparse.FLANgrid method*), 52
_get_target_ordering() (*deeprank.learn.DataSet.DataSet method*), 31
_get_variables() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_get_vdw() (*deeprank.features.AtomicFeature.AtomicFeature method*), 42
_init_conv_layer_random() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
_init_fc_layer_random() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
_load() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
_mad_based_outliers() (*deeprank.learn.DataSet.DataSet static method*), 31
_normalize_feature() (*deeprank.learn.DataSet.DataSet method*), 31
_normalize_target() (*deeprank.learn.DataSet.DataSet method*), 31
_plot_boxplot_class() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_plot_scatter_reg() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
_prefactor_vdw() (*deeprank.features.AtomicFeature.AtomicFeature static method*), 43
_printif() (*in module deeprank.generate.DataGenerator*), 15
_printif() (*in module deeprank.tools.sparse*), 51
_process_data() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
_read_norm() (*deeprank.learn.DataSet.DataSet method*), 31
_rotate_feature() (*deeprank.generate.DataGenerator.DataGenerator static method*), 22
_select_pdb() (*deeprank.learn.DataSet.DataSet method*), 29
_test_cuda() (*deeprank.generate.DataGenerator.DataGenerator method*), 20
_train() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
_tunable_kernel() (*deeprank.generate.DataGenerator.DataGenerator static method*), 21
_tune_cuda_kernel() (*deeprank.generate.DataGenerator.DataGenerator method*), 20
_write_conv_output() (*deeprank.learn.modelGenerator.NetworkGenerator static method*), 38
_write_definition() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 38
_write_forward() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
_write_forward_feature() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
_write_import() (*deeprank.learn.modelGenerator.NetworkGenerator static method*), 38
_write_init() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 38

A

add() (*deeprank.generate.NormalizeData.NormParam method*), 24
add_all_atomic_densities() (*deeprank.generate.GridTools.GridTools method*), 25
add_all_features() (*deeprank.generate.GridTools.GridTools method*),

25
add_feature() (deeprank.generate.DataGenerator.DataGenerator method), 18
add_target() (deeprank.generate.DataGenerator.DataGenerator method), 18
add_unique_target() (deeprank.generate.DataGenerator.DataGenerator method), 18
assign_parameters() (deeprank.features.AtomicFeature.AtomicFeature method), 42
AtomicFeature (class in deeprank.features.AtomicFeature), 40
aug_data() (deeprank.generate.DataGenerator.DataGenerator method), 17

B

backtransform_target() (deeprank.learn.DataSet.DataSet method), 31
BSA (class in deeprank.features.BSA), 43

C

check_hdf5_files() (deeprank.learn.DataSet.DataSet static method), 29
compute_coulomb_interchain_only() (deeprank.features.AtomicFeature.AtomicFeature method), 42
compute_norm() (deeprank.learn.DataSet.DataSet method), 31
compute_vdw_interchain_only() (deeprank.features.AtomicFeature.AtomicFeature method), 42
conv (class in deeprank.learn.modelGenerator), 39
convert2d() (deeprank.learn.DataSet.DataSet static method), 32
convertSeconds2Days() (deeprank.learn.NeuralNet.NeuralNet static method), 35
create_database() (deeprank.generate.DataGenerator.DataGenerator method), 17
create_index_molecules() (deeprank.learn.DataSet.DataSet method), 29
create_new_data() (deeprank.generate.GridTools.GridTools method), 25

D

DataGenerator (class in deeprank.generate.DataGenerator), 15
DataSet (class in deeprank.learn.DataSet), 27
deeprank.features.AtomicFeature (module), 40
deeprank.features.BSA (module), 43
deeprank.features.FeatureClass (module), 46
deeprank.features.FullPSSM (module), 44
deeprank.features.PSSM_IC (module), 45
deeprank.features.ResidueDensity (module), 46
deeprank.generate (module), 13
deeprank.generate.DataGenerator (module), 15
deeprank.generate.GridTools (module), 24
deeprank.generate.NormalizeData (module), 23
deeprank.learn (module), 26
deeprank.learn.DataSet (module), 27
deeprank.learn.modelGenerator (module), 38
deeprank.learn.NeuralNet (module), 34
deeprank.targets (module), 47
deeprank.tools (module), 49
deeprank.tools.sasa (module), 49
deeprank.tools.sparse (module), 51
define_grid_points() (deeprank.generate.GridTools.GridTools method), 25
densgrid() (deeprank.generate.GridTools.GridTools method), 25
dropout (class in deeprank.learn.modelGenerator), 39

E

evaluate_charges() (deeprank.features.AtomicFeature.AtomicFeature method), 42
evaluate_pair_interaction() (deeprank.features.AtomicFeature.AtomicFeature method), 42
export_data_hdf5() (deeprank.features.FeatureClass.FeatureClass method), 47
export_dataxyz_hdf5() (deeprank.features.FeatureClass.FeatureClass method), 47
export_grid_points() (deeprank.generate.GridTools.GridTools method), 26
extract_features() (deeprank.features.ResidueDensity.ResidueDensity method), 46

F

fc (class in deeprank.learn.modelGenerator), 39
featgrid() (deeprank.generate.GridTools.GridTools method), 26

FeatureClass (*class in deeprank.features.FeatureClass*), 46
filter() (*deeprank.learn.DataSet.DataSet method*), 29
FLANgrid (*class in deeprank.tools.sparse*), 51
from_dense() (*deeprank.tools.sparse.FLANgrid method*), 51
FullPSSM (*class in deeprank.features.FullPSSM*), 44

G

get() (*deeprank.features.ResidueDensity.ResidueDensity method*), 46
get() (*deeprank.generate.NormalizeData.NormalizeData method*), 23
get_center() (*deeprank.tools.sasa.SASA method*), 50
get_contact_atoms() (*deeprank.features.AtomicFeature.AtomicFeature method*), 41
get_contact_center() (*deeprank.generate.GridTools.GridTools method*), 25
get_contact_residue_sasa() (*deeprank.features.BSA.BSA method*), 43
get_feature_value() (*deeprank.features.FullPSSM.FullPSSM method*), 44
get_grid() (*deeprank.learn.DataSet.DataSet method*), 32
get_grid_shape() (*deeprank.learn.DataSet.DataSet method*), 31
get_input_shape() (*deeprank.learn.DataSet.DataSet method*), 30
get_mapped_feature_name() (*deeprank.learn.DataSet.DataSet method*), 30
get_new_random_model() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
get_norm() (*deeprank.learn.DataSet.DataSet method*), 31
get_pairing_feature() (*deeprank.learn.DataSet.DataSet method*), 30
get_raw_feature_name() (*deeprank.learn.DataSet.DataSet method*), 30
get_ref_mol_name() (*deeprank.features.FullPSSM.FullPSSM static method*), 44
get_residue_carbon_beta() (*deeprank.tools.sasa.SASA method*), 50
get_residue_center() (*deeprank.features.FeatureClass.FeatureClass static method*), 47
get_residue_center() (*deeprank.tools.sasa.SASA method*), 50
get_structure() (*deeprank.features.BSA.BSA method*), 43
GridTools (*class in deeprank.generate.GridTools*), 24

H

hdf5_grid_data() (*deeprank.generate.GridTools.GridTools method*), 26

L

load_data_params() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
load_model_params() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
load_nn_params() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
load_one_molecule() (*deeprank.learn.DataSet.DataSet method*), 32
load_optimizer_params() (*deeprank.learn.NeuralNet.NeuralNet method*), 36
logif() (*in module deeprank.generate.GridTools*), 24

M

make_feature_pair() (*deeprank.learn.DataSet.DataSet static method*), 32
map_atomic_densities() (*deeprank.generate.GridTools.GridTools method*), 25
map_atomic_densities() (*deeprank.learn.DataSet.DataSet method*), 33
map_feature() (*deeprank.learn.DataSet.DataSet method*), 33
map_features() (*deeprank.generate.DataGenerator.DataGenerator method*), 19
map_features() (*deeprank.generate.GridTools.GridTools method*), 26
map_one_molecule() (*deeprank.learn.DataSet.DataSet method*), 32
MinMaxParam (*class in deeprank.generate.NormalizeData*), 24

N

neighbor_count() (*deeprank.tools.sasa.SASA method*), 50
neighbor_vector() (*deeprank.tools.sasa.SASA method*), 50

neighbor_weight() (*deeprank.tools.sasa.SASA static method*), 51
NetworkGenerator (*class in deeprank.learn.modelGenerator*), 38
NeuralNet (*class in deeprank.learn.NeuralNet*), 34
NormalizeData (*class in deeprank.generate.NormalizeData*), 23
NormParam (*class in deeprank.generate.NormalizeData*), 23

P

plot_hit_rate() (*deeprank.learn.NeuralNet.NeuralNet method*), 37
pool (*class in deeprank.learn.modelGenerator*), 39
precompute_grid() (*deeprank.generate.DataGenerator.DataGenerator method*), 19
print() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 39
print_possible_features() (*deeprank.learn.DataSet.DataSet method*), 30
process() (*deeprank.generate.NormalizeData.NormParam method*), 24
process_dataset() (*deeprank.learn.DataSet.DataSet method*), 29
PSSM_IC (*class in deeprank.features.PSSM_IC*), 45

R

read_charge_file() (*deeprank.features.AtomicFeature.AtomicFeature method*), 41
read_patch() (*deeprank.features.AtomicFeature.AtomicFeature method*), 41
read_pdb() (*deeprank.generate.GridTools.GridTools method*), 25
read_PSSM_data() (*deeprank.features.FullPSSM.FullPSSM method*), 44
read_vdw_file() (*deeprank.features.AtomicFeature.AtomicFeature method*), 41
realign_complexes() (*deeprank.generate.DataGenerator.DataGenerator method*), 19
remove() (*deeprank.generate.DataGenerator.DataGenerator method*), 20
residue_pair (*class in deeprank.features.ResidueDensity*), 46
ResidueDensity (*class in deeprank.features.ResidueDensity*), 46

S

SASA (*class in deeprank.tools.sasa*), 49
save_model() (*deeprank.learn.NeuralNet.NeuralNet method*), 36

T

test() (*deeprank.learn.NeuralNet.NeuralNet method*), 35
to_dense() (*deeprank.tools.sparse.FLANgrid method*), 51
train() (*deeprank.learn.NeuralNet.NeuralNet method*), 35

U

update() (*deeprank.generate.NormalizeData.MinMaxParam method*), 24
update_feature() (*deeprank.generate.GridTools.GridTools method*), 25

W

write() (*deeprank.learn.modelGenerator.NetworkGenerator method*), 38