DeepRank-GNN Documentation

Release 0.1.0

Manon Reau, Nicolas Renaud

Dec 06, 2022

DEEPRANK-GNN:

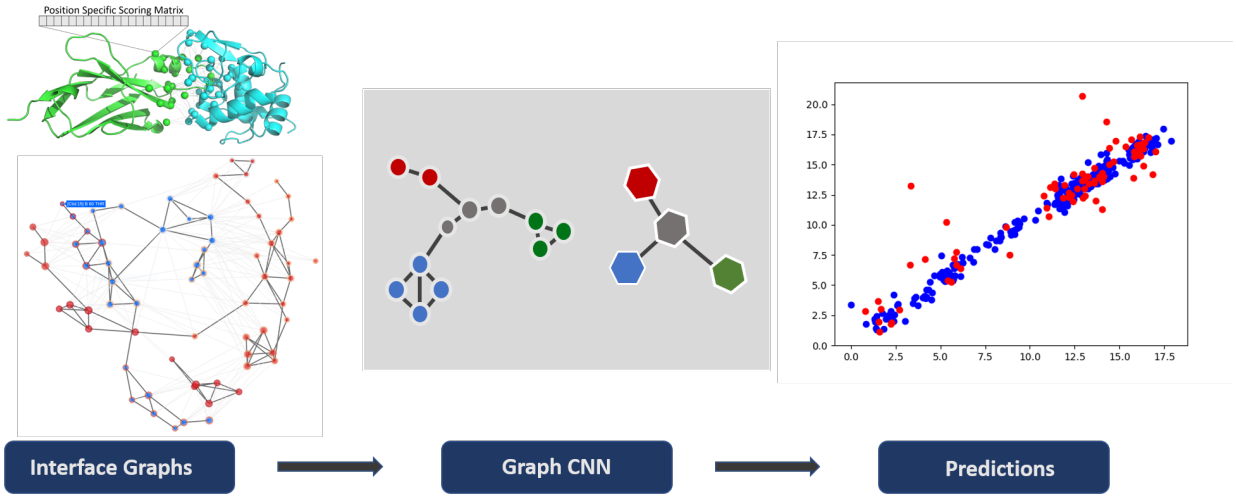
CHAPTER

ONE

OVERVIEW

DeepRank-GNN is a framework that converts PPI interfaces into graphs and uses those to learn interaction patterns using Graph Neural Networks.

The framework is designed to be adapted to any PPI related research project.



Deeprank-GNN works in a two step process:

1.1 1) Graph generation

Use your own protein-protein complexes to generate PPI interface graphs.

Go to *Creating Graphs*

1.2 2) Model Training

Tune and train your Graph Neural Network and make your own predictions.

Go to *Training a module*

CHAPTER

TWO

MOTIVATIONS

Protein-protein interactions (PPIs) are essential in all cellular processes of living organisms including cell growth, structure, communication, protection and death. Acquiring knowledge on PPI is fundamental to understand normal and altered physiological processes and propose solutions to restore them. In the past decades, a large number of PPI structures have been solved by experimental approaches (e.g., X-ray crystallography, nuclear magnetic resonance, cryogenic electron microscopy). Given the remarkable success of Convolutional Neural Network (CNN) in retrieving patterns in images[1], CNN architectures have been developed to learn interaction patterns in PPI interfaces[2],[3].

CNNs however come with major limitations: First, they are sensitive to the input PPI orientation, which may require data augmentation (i.e. multiple rotations of the input data) for the network to forget about the orientation in the learning process; second, the size of the 3D grid is unique for all input data, which does not reflect the variety in interface sizes observed in experimental structures and may be problematic for large interfaces that do not fit inside the predefined grid size. A solution to this problem is to use instead Graph Neural networks (GNN). By definition, graphs are non-structured geometric structures and do not hold orientation information. They are rotational invariant and can easily represent interfaces of varying sizes.

Building up on our previous tool DeepRank(https://github.com/DeepRank/deeprank) that maps atomic and residue-level features from PPIs to 3D grids and applies 3D CNNs to learn problem-specific interaction patterns, we present here Deeprank-GNN. Deeprank-GNN converts PPI interfaces into graphs and uses those to learn interaction patterns.

DeepRank-GNN is a framework than can be easily used by the community and adapted to any topic involving PPI interactions. The framework allows users to define their own graph neural network, features and target values.

[1] Krizhevsky A, Sutskever I, Hinton GE, ImageNet classification with deep convolutional neural networks. Adv Neural Inf Process Syst 25, 2012

[2] Renaud N, Geng C, Georgievska S, Ambrosetti F, Ridder L, Marzella D, Bonvin A, Xue L, DeepRank: A deep learning framework for data mining 3D protein-protein interfaces, bioRxiv, 2021.01.29.425727

[3] Wang X, Terashi G, Christoffer CW, Zhu M, Kihara D. Protein docking model evaluation by 3D deep convolutional neural networks. Bioinformatics. 2020 ;36(7):2113-2118.

CHAPTER

THREE

INSTALLATION

3.1 Via Python Package

The latest release of DeepRank-GNN can be installed using the pypi package manager with :

`pip install deeprank-gnn`

If you are planning to only use DeepRank-GNN this is the quickest way to obtain the code

3.2 Via GitHub

For user who would like to contribute, the code is hosted on GitHub (https://github.com/DeepRank/DeepRank-GNN)

To install the code

- clone the repository `git clone https://github.com/DeepRank/DeepRank-GNN.git`
- go there `cd DeepRank-GNN`
- install the module `pip install -e ./`

You can then test the installation :

- `cd test`
- `pytest`

Note: Ensure that at least PyTorch 1.5.0 is installed:

`python -c "import torch; print(torch.__version__)`

In case PyTorch Geometric installation fails, refer to the installation guide: https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html

---

CHAPTER

FOUR

CREATING GRAPHS

Deeprank-GNN automatically generates a residue-level graphs of **protein-protein interfaces** in which nodes correspond to a single residue, and 2 types of edges are defined:

- **External edges** connect 2 residues (nodes) of chain A and B if they have at least 1 pairwise atomic distance **< 8.5 A** (Used for to define neighbors)
- **Internal edges** connect 2 residues (nodes) within a chain if they have at least 1 pairwise atomic distance **< 3 A** (Used to cluster nodes)

Warning: The graph generation requires an ensemble of PDB files containing two chains: chain **A** and chain **B**.

You can provide PSSM matrices to compute evolutionary conservation node features. Some pre-calculated PSSM matrices can be downloaded from http://3dcons.cnb.csic.es/. A `3dcons_to_deeprank_pssm.py` converter can be found in the `tool` folder to convert the 3dcons PSSM format into the Deeprank-GNN PSSM format. **Make sure the sequence numbering matches the PDB residues numbering.**

By default, the following features are assigned to each node of the graph :

- **pos**: xyz coordinates
- **chain**: chain ID
- **charge**: residue charge
- **polarity**: apolar/polar/neg_charged/pos_charged (one hot encoded)
- **bsa**: buried surface are
- **type**: residue type (one hot encoded)

The following features are computed if PSSM data is provided :

- **pssm**: pssm score for each residues
- **cons**: pssm score of the residue
- **ic**: information content of the PSSM (~Shannon entropy)

The following features are optional, and computed only if Biopython is used (see next example) :

- **depth**: average atom depth of the atoms in a residue (distance to the surface)
- **hse**: half sphere exposure

4.1 Generate your graphs

Note that the pssm information is used to compute the **pssm**, **cons** and **ic** node features and is optional.

In this example, all features are computed

```
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>>
>>> pdb_path = './data/pdb/1ATN/'
>>> pssm_path = './data/pssm/1ATN/'
>>>
>>> GraphHDF5(pdb_path=pdb_path, pssm_path=pssm_path, biopython=True,
>>>           graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
```

In this example, the biopython features (hse and depth) are ignored

```
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>>
>>> pdb_path = './data/pdb/1ATN/' # path to the docking model in PDB format
>>> pssm_path = './data/pssm/1ATN/' # path to the pssm files
>>>
>>> GraphHDF5(pdb_path=pdb_path, pssm_path=pssm_path,
>>>           graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
```

In this example, the biopython features (hse and depth) and the PSSM information are ignored

```
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>>
>>> pdb_path = './data/pdb/1ATN/'
>>>
>>> GraphHDF5(pdb_path=pdb_path,
>>>           graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
```

4.2 Add your target values

Use the CustomizeGraph class to add target values to the graphs.

If you are benchmarking docking models, go to the **next section**.

```
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>> from deeprank_gnn.tools.CustomizeGraph import add_target
>>>
>>> pdb_path = './data/pdb/1ATN/'
>>> pssm_path = './data/pssm/1ATN/'
>>>
>>> GraphHDF5(pdb_path=pdb_path, pssm_path=pssm_path,
>>>           graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
>>>
>>> add_target(graph_path='.', target_name='new_target',
>>>            target_list='list_of_target_values.txt')
```

Note: The list of target values should respect the following format:

```
model_name_1 0
model_name_2 1
model_name_3 0
model_name_4 0
```

if your use other separators (eg. `,`, `;`, `tab`) use the `sep` argument:

```
>>> add_target(graph_path=graph_path, target_name='new_target',
>>>            target_list='list_of_target_values.txt', sep=',')
```

4.3 Docking benchmark mode

In a docking benchmark mode, you can provide the path to the reference structures in the graph generation step. Knowing the reference structure, the following target values will be automatically computed, based on CAPRI quality criteria[1], and assigned to the graphs :

- **irmsd**: interface RMSD (RMSD between the superimposed interface residues)
- **lrmsd**: ligand RMSD (RMSD between chains B given that chains A are superimposed)
- **fnat**: fraction of native contacts
- **dockQ**: see Basu et al., "DockQ: A Quality Measure for Protein-Protein Docking Models", PLOS ONE, 2016
- **bin_class**: binary classification (0: `irmsd >= 4 A`, 1: `RMSD < 4A`)
- **capri_classes**: 1: `RMSD < 1A`, 2: `RMSD < 2A`, 3: `RMSD < 4A`, 4: `RMSD < 6A`, 0: `RMSD >= 6A`

```
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>>
>>> pdb_path = './data/pdb/1ATN/'
>>> pssm_path = './data/pssm/1ATN/'
>>> ref = './data/ref/1ATN/'
>>>
>>> GraphHDF5(pdb_path=pdb_path, ref_path=ref, pssm_path=pssm_path,
>>>           graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
```

Note: The different input files must respect the following nomenclature:

- PDB files: `1ATN_xxx.pdb` (xxx may be replaced by anything)
- PSSM files: `1ATN.A.pdb.pssm 1ATN.B.pdb.pssm` or `1ATN.A.pssm 1ATN.B.pssm`
- Reference PDB files: `1ATN.pdb`

[1] Lensink MF, Méndez R, Wodak SJ, Docking and scoring protein complexes: CAPRI 3rd Edition. Proteins. 2007

CHAPTER

FIVE

TRAINING A MODULE

5.1 1. Select node and edge features

5.1.1 1.1. Edge feature:

- **dist**: distance between nodes

```
>>> edge_feature=['dist']
```

Note: **External edges** connect 2 residues of chain A and B if they have at least 1 pairwise atomic distance **< 8.5 A** (Used for to define neighbors)

Internal edges connect 2 residues within a chain if they have at least 1 pairwise atomic distance **< 3 A** (Used to cluster nodes)

5.1.2 1.2. Node features:

- **pos**: xyz coordinates
- **chain**: chain ID
- **charge**: residue charge
- **polarity**: apolar/polar/neg_charged/pos_charged (one hot encoded)
- **bsa**: buried surface are
- **pssm**: pssm score for each residues
- **cons**: pssm score of the residue
- **ic**: information content of the PSSM (~Shannon entropy)
- **type**: residue type (one hot encoded)
- **depth** (opt. in graph. gen. step): average atom depth of the atoms in a residue (distance to the surface)
- **hse** (opt. in graph. gen. step): half sphere exposure

```
>>> node_feature=['type', 'polarity', 'bsa',
>>>               'ic', 'pssm']
```

5.2 2. Select the target (benchmarking mode)

When using Deeprank-GNN in a benchmarking mode, you must specify your target (often referred to as Y). The target values are pre-calculated during the Graph generation step **if a reference structure is provided**.

Pre-calculated targets:

- **irmsd**: interface RMSD (RMSD between the superimposed interface residues)
- **lrmsd**: ligand RMSD (RMSD between chains B given that chains A are superimposed)
- **fnat**: fraction of native contacts
- **dockQ**: see Basu et al., "DockQ: A Quality Measure for Protein-Protein Docking Models", PLOS ONE, 2016
- **bin_class**: binary classification (0: `irmsd >= 4 A`, 1: `RMSD < 4A`)
- **capri_classes**: 1: `RMSD < 1A`, 2: `RMSD < 2A`, 3: `RMSD < 4A`, 4: `RMSD < 6A`, 0: `RMSD >= 6A`

Note: In classification mode (i.e. task="class") you must provide the list of target classes to the NeuralNet (e.g. classes=[1,2,3,4])

```
>>> target='irmsd'
```

5.3 3. Select hyperparameters

- Regression ('reg') of classification ('class') mode

```
>>> task='reg'
```

- Batch size

```
>>> batch_size=64
```

- Shuffle the training dataset

```
>>> shuffle=True
```

- Learning rate:

```
>>> lr=0.001
```

5.4 4. Load the network

This step requires pre-calculated graphs in hdf5 format.

The user may :

- option 1: input a unique dataset and chose to automatically split it into a training set and an evaluation set
- option 2: input distinct training/evaluation/test sets

5.4.1 4.1. Option 1

```
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>>
>>> database = './1ATN_residue.hdf5'
>>>
>>> model = NeuralNet(database, GINet,
>>>                node_feature=node_feature,
>>>                edge_feature=edge_feature,
>>>                target=target,
>>>                task=task,
>>>                lr=lr,
>>>                batch_size=batch_size,
>>>                shuffle=shuffle,
>>>                percent=[0.8, 0.2])
>>>
```

Note: The *percent* argument is required to split the input dataset into a training set and a test set. Using `percent=[0.8, 0.2]`, 80% of the input dataset will constitute the training set, 20% will constitute the evaluation set.

5.4.2 4.2. Option 2

```
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>> import glob
>>>
>>> # load train dataset
>>> database_train = glob.glob('./hdf5/train*.hdf5')
>>> # load validation dataset
>>> database_eval = glob.glob('./hdf5/eval*.hdf5')
>>> # load test dataset
>>> database_test = glob.glob('./hdf5/test*.hdf5')
>>>
>>> model = NeuralNet(database_train, GINet,
>>>                node_feature=node_feature,
>>>                edge_feature=edge_attr,
>>>                target=target,
>>>                task=task,
>>>                lr=lr,
>>>                batch_size=batch_size,
>>>                shuffle=shuffle,
>>>                database_eval = database_eval)
```

5.5 5. Train the model

- example 1:

train the network, perform 50 epochs

```
>>> model.train(nepoch=50, validate=False)
```

- example 2:

train the model, evaluate the model at each epoch, save the best model (i.e. the model with the lowest loss), and write all predictions to `output.hdf5`

```
>>> model.train(nepoch=50, validate=True, save_model='best', hdf5='output.hdf5')
```

Warning: The `last` model is saved by default.

When setting `save_model='best'`, a model that is associated with a lower loss than those generated in the previous epochs will be saved. By default, the epoch number is included in the output name not to write over intermediate models.

5.6 6. Analysis

5.6.1 6.1. Plot the loss evolution over the epochs

```
>>> model.plot_loss(name='plot_loss')
```

5.6.2 6.2 Analyse the performance in benchmarking conditions

The following analysis only apply if a reference structure was provided during the graph generation step.

6.2.1. **Plot accuracy evolution**

```
>>> model.plot_acc(name='plot_accuracy')
```

6.2.2. **Plot hitrate**

A threshold value is required to binarise the target value

```
>>> model.plot_hit_rate(data='eval', threshold=4.0, mode='percentage', name='hitrate_eval
↪')
```

6.2.3. **Get various metrics**

The following metrics can be easily computed:

Classification metrics:

- **sensitivity**: Sensitivity, hit rate, recall, or true positive rate
- **specificity**: Specificity or true negative rate
- **precision**: Precision or positive predictive value

- **NPV**: Negative predictive value
- **FPR**: Fall out or false positive rate
- **FNR**: False negative rate
- **FDR**: False discovery rate
- **accuracy**: Accuracy
- **auc**(): AUC
- **hitrate**(): Hit rate

Regression metrics:

- **explained_variance**: Explained variance regression score function
- **max_error**: Max_error metric calculates the maximum residual error
- **mean_abolute_error**: Mean absolute error regression loss
- **mean_squared_error**: Mean squared error regression loss
- **root_mean_squared_error**: Root mean squared error regression loss
- **mean_squared_log_error**: Mean squared logarithmic error regression loss
- **median_squared_log_error**: Median absolute error regression loss
- **r2_score**: R^2 (coefficient of determination) regression score function

---

Note: All classification metrics can be calculated on continuous targets as soon as a threshold is provided to binarise the data.

```
>>> train_metrics = model.get_metrics('train', threshold = 4.0)
>>> print('training set - accuracy:', train_metrics.accuracy)
>>> print('training set - sensitivity:', train_metrics.sensitivity)
>>>
>>> eval_metrics = model.get_metrics('eval', threshold = 4.0)
>>> print('evaluation set - accuracy:', eval_metrics.accuracy)
>>> print('evaluation set - sensitivity:', eval_metrics.sensitivity)
```

5.7 7. Save the model/network

```
>>> model.save_model("model_backup.pth.tar")
```

5.8 8. Test the model on an external dataset

5.8.1 8.1. On a loaded model

```
>>> model.test(database_test, threshold=4.0)
```

5.8.2 8.2. On a pre-trained model

```
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>>
>>> database_test = './1ATN_residue.hdf5'
>>>
>>> model = NeuralNet(database_test, GINet, pretrained_model = "model_backup.pth.tar")
>>> model.test(database_test)
>>>
>>> test_metrics = model.get_metrics('test', threshold = 4.0)
>>> print(test_metrics.accuracy)
```

CHAPTER

SIX

IN SHORT

```
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>>
>>> database = './1ATN_residue.hdf5'
>>>
>>> edge_feature=['dist']
>>> node_feature=['type', 'polarity', 'bsa',
>>>               'depth', 'hse', 'ic', 'pssm']
>>> target='irmsd'
>>> task='reg'
>>> batch_size=64
>>> shuffle=True
>>> lr=0.001
>>>
>>> model = NeuralNet(database, GINet,
>>>                node_feature=node_feature,
>>>                edge_feature=edge_feature,
>>>                target=target,
>>>                index=None,
>>>                task=task,
>>>                lr=lr,
>>>                batch_size=batch_size,
>>>                shuffle=shuffle,
>>>                percent=[0.8, 0.2])
>>>
>>> model.train(nepoch=50, validate=True, save_model='best', hdf5='output.hdf5')
>>> model.plot_loss(name='plot_loss')
>>>
>>> train_metrics = model.get_metrics('train', threshold = 4.0)
>>> print('training set - accuracy:', train_metrics.accuracy)
>>> print('training set - sensitivity:', train_metrics.sensitivity)
>>>
>>> eval_metrics = model.get_metrics('eval', threshold = 4.0)
>>> print('evaluation set - accuracy:', eval_metrics.accuracy)
>>> print('evaluation set - sensitivity:', eval_metrics.sensitivity)
>>>
>>> model.save_model("model_backup.pth.tar")
>>> #model.test(database_test, threshold=4.0)
```

Using default settings

```
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>>
>>> database = glob.glob('./hdf5/*_train.hdf5')
>>> dataset_test = glob.glob('./hdf5/*_test.hdf5')
>>>
>>> target='irmsd'
>>>
>>> model = NeuralNet(database, GINet,
>>>                target=target,
>>>                percent=[0.8, 0.2])
>>>
>>> model.train(nepoch=50, validate=True, save_model='best', hdf5='output.hdf5')
>>> model.plot_loss(name='plot_loss')
>>>
>>> train_metrics = model.get_metrics('train', threshold = 4.0)
>>> print('training set - accuracy:', train_metrics.accuracy)
>>> print('training set - sensitivity:', train_metrics.sensitivity)
>>>
>>> eval_metrics = model.get_metrics('eval', threshold = 4.0)
>>> print('evaluation set - accuracy:', eval_metrics.accuracy)
>>> print('evaluation set - sensitivity:', eval_metrics.sensitivity)
>>>
>>> model.save_model("model_backup.pth.tar")
>>> model.test(database_test, threshold=4.0)
```

CHAPTER

SEVEN

USE DEEPRANK-GNN PAPER'S PRETRAINED MODEL

See: M. Réau, N. Renaud, L. C. Xue, A. M. J. J. Bonvin, "DeepRank-GNN: A Graph Neural Network Framework to Learn Patterns in Protein-Protein Interfaces", bioRxiv 2021.12.08.471762; doi: https://doi.org/10.1101/2021.12.08.471762

You can get the pre-trained model from DeepRank-GNN github repository

```
>>> import glob
>>> import sys
>>> import time
>>> import datetime
>>> import numpy as np
>>>
>>> from deeprank_gnn.GraphGenMP import GraphHDF5
>>> from deeprank_gnn.NeuralNet import NeuralNet
>>> from deeprank_gnn.ginet import GINet
>>>
>>> ### Graph generation section
>>> pdb_path = '../tests/data/pdb/1ATN/'
>>> pssm_path = '../tests/data/pssm/1ATN/'
>>>
>>> GraphHDF5(pdb_path=pdb_path, pssm_path=pssm_path,
>>>         graph_type='residue', outfile='1ATN_residue.hdf5', nproc=4)
>>>
>>> ### Prediction section
>>> gnn = GINet
>>> pretrained_model = 'fold6_treg_yfnat_b128_e20_lr0.001_4.pt'
>>> database_test = glob.glob('1ATN_residue.hdf5')
>>>
>>> start_time = time.time()
>>> model = NeuralNet(database_test, gnn, pretrained_model = pretrained_model)
>>> model.test(threshold=None)
>>> end_time = time.time()
>>> print ('Elapsed time: {end_time-start_time}')
>>>
>>> ### The output is automatically stored in **test_data.hdf5**
```

Note: For storage convenience, all predictions are stored in a HDF5 file. A converter from HDF5 to csv is provided in the **tools** directory

CHAPTER

EIGHT

ADVANCED

8.1 Design your own GNN layer

Example:

```
>>> import torch
>>> from torch.nn import Parameter
>>> import torch.nn.functional as F
>>> import torch.nn as nn
>>>
>>> from torch_scatter import scatter_mean
>>> from torch_scatter import scatter_sum
>>>
>>> from torch_geometric.utils import remove_self_loops, add_self_loops, softmax
>>>
>>> # torch_geometric import
>>> from torch_geometric.nn.inits import uniform
>>> from torch_geometric.nn import max_pool_x
>>>
>>> class GINet_layer(torch.nn.Module):
>>>
>>>     def __init__(self,
>>>                  in_channels,
>>>                  out_channels,
>>>                  number_edge_features=1,
>>>                  bias=False):
>>>
>>>         super(EGAT, self).__init__()
>>>
>>>         self.in_channels = in_channels
>>>         self.out_channels = out_channels
>>>
>>>         self.fc = nn.Linear(self.in_channels, self.out_channels, bias=bias)
>>>         self.fc_edge_attr = nn.Linear(number_edge_features, 3, bias=bias)
>>>         self.fc_attention = nn.Linear(2 * self.out_channels + 3, 1, bias=bias)
>>>         self.reset_parameters()
>>>
>>>     def reset_parameters(self):
>>>
>>>         size = self.in_channels
```

(continues on next page)

(continued from previous page)

```
>>>         uniform(size, self.fc.weight)
>>>         uniform(size, self.fc_attention.weight)
>>>         uniform(size, self.fc_edge_attr.weight)
>>>
>>>     def forward(self, x, edge_index, edge_attr):
>>>
>>>         row, col = edge_index
>>>         num_node = len(x)
>>>         edge_attr = edge_attr.unsqueeze(
>>>             -1) if edge_attr.dim() == 1 else edge_attr
>>>
>>>         xcol = self.fc(x[col])
>>>         xrow = self.fc(x[row])
>>>
>>>         ed = self.fc_edge_attr(edge_attr)
>>>         # create edge feature by concatenating node features
>>>         alpha = torch.cat([xrow, xcol, ed], dim=1)
>>>         alpha = self.fc_attention(alpha)
>>>         alpha = F.leaky_relu(alpha)
>>>
>>>         alpha = F.softmax(alpha, dim=1)
>>>         h = alpha * xcol
>>>
>>>         out = torch.zeros(
>>>             num_node, self.out_channels).to(alpha.device)
>>>         z = scatter_sum(h, row, dim=0, out=out)
>>>
>>>         return z
>>>
>>>     def __repr__(self):
>>>         return '{}({}, {})'.format(self.__class__.__name__,
>>>                                    self.in_channels,
>>>                                    self.out_channels)
```

8.2 Design your own neural network architecture

The provided example makes use of **internal edges** and **external edges**.

We perform convolutions on the internal and external edges independently by providing the following data to the convolution layers:

- data_ext.internal_edge_index, data_ext.internal_edge_attr for the **internal** edges
- data_ext.edge_index, data_ext.edge_attr for the **external** edges

The GNN class **must** be initialized with 3 arguments: - input_shape - output_shape - input_shape_edge

```
>>> class GINet(torch.nn.Module):
>>>     def __init__(self, input_shape, output_shape = 1, input_shape_edge = 1):
>>>         super(GINet_layer, self).__init__()
>>>         self.conv1 = GINet_layer(input_shape, 16)
>>>         self.conv2 = GINet_layer(16, 32)
```

(continues on next page)

(continued from previous page)

```
>>>
>>>         self.conv1_ext = GINet_layer(input_shape, 16, input_shape_edge)
>>>         self.conv2_ext = GINet_layer(16, 32, input_shape_edge)
>>>
>>>         self.fc1 = nn.Linear(2*32, 128)
>>>         self.fc2 = nn.Linear(128, output_shape)
>>>         self.clustering = 'mcl'
>>>         self.dropout = 0.4
>>>
>>>     def forward(self, data):
>>>         act = F.relu
>>>         data_ext = data.clone()
>>>
>>>         # INTER-PROTEIN INTERACTION GRAPH
>>>         # first conv block
>>>         data.x = act(self.conv1(
>>>             data.x, data.edge_index, data.edge_attr))
>>>         cluster = get_preloaded_cluster(data.cluster0, data.batch)
>>>         data = community_pooling(cluster, data)
>>>
>>>         # second conv block
>>>         data.x = act(self.conv2(
>>>             data.x, data.edge_index, data.edge_attr))
>>>         cluster = get_preloaded_cluster(data.cluster1, data.batch)
>>>         x, batch = max_pool_x(cluster, data.x, data.batch)
>>>
>>>         # INTRA-PROTEIN INTERACTION GRAPH
>>>         # first conv block
>>>         data_ext.x = act(self.conv1_ext(
>>>             data_ext.x, data_ext.internal_edge_index, data_ext.internal_edge_attr))
>>>         cluster = get_preloaded_cluster(data_ext.cluster0, data_ext.batch)
>>>         data_ext = community_pooling(cluster, data_ext)
>>>
>>>         # second conv block
>>>         data_ext.x = act(self.conv2_ext(
>>>             data_ext.x, data_ext.internal_edge_index, data_ext.internal_edge_attr))
>>>         cluster = get_preloaded_cluster(data_ext.cluster1, data_ext.batch)
>>>         x_ext, batch_ext = max_pool_x(cluster, data_ext.x, data_ext.batch)
>>>
>>>         # FC
>>>         x = scatter_mean(x, batch, dim=0)
>>>         x_ext = scatter_mean(x_ext, batch_ext, dim=0)
>>>
>>>         x = torch.cat([x, x_ext], dim=1)
>>>         x = act(self.fc1(x))
>>>         x = F.dropout(x, self.dropout, training=self.training)
>>>         x = self.fc2(x)
>>>
>>>         return x
```

8.3 Use your GNN architecture in Deeprank-GNN

```
>>> model = NeuralNet(database, GINet,
>>>                   node_feature=node_feature,
>>>                   edge_feature=edge_feature,
>>>                   target=target,
>>>                   task=task,
>>>                   lr=lr,
>>>                   batch_size=batch_size,
>>>                   shuffle=shuffle,
>>>                   percent=[0.8, 0.2])
```

CHAPTER

NINE

GRAPH AND GRAPH GENERATOR

9.1 Graphs

class deeprank_gnn.Graph.**Graph**

Class that performs graph level action

- get score for the graph (lrmsd, irmsd, fnat, capri_class, bin_class and dockQ)
- networkx object (graph) to hdf5 format
- networkx object (graph) from hdf5 format

get_score(*ref*)

Assigns scores (lrmsd, irmsd, fnat, dockQ, bin_class, capri_class) to a protein graph

Parameters

ref (*path*) – path to the reference structure required to compute the different score

nx2h5(*f5*)

Converts Networkx object to hdf5 format

Parameters

f5 (*[type]*) – hdf5 file

h52nx(*f5name*, *mol*, *molgrp=None*)

Converts Hdf5 file to Networkx object

Parameters

- **f5name** (*str*) – hdf5 file
- **mol** (*str*) – molecule name
- **molgrp** (*[type], optional*) – hdf5[molecule]. Defaults to None.

plotly_2d(*out=None*, *offline=False*, *iplot=True*, *disable_plot=False*, *method='louvain'*)

Plots the interface graph in 2D

Parameters

- **out** (*[type], optional*) – output name. Defaults to None.
- **offline** (*bool, optional*) – Defaults to False.
- **iplot** (*bool, optional*) – Defaults to True.
- **method** (*str, optional*) – ‘mcl’ of ‘louvain’. Defaults to ‘louvain’.

plotly_3d(*out=None*, *offline=False*, *iplot=True*, *disable_plot=False*)

Plots interface graph in 3D

Parameters

- **out** (*[type], optional*) – [description]. Defaults to None.
- **offline** (*bool, optional*) – [description]. Defaults to False.
- **iplot** (*bool, optional*) – [description]. Defaults to True.

9.2 Residue Graphs

class deeprank_gnn.ResidueGraph.**ResidueGraph**(*pdb=None*, *pssm=None*, *contact_distance=8.5*, *internal_contact_distance=3*, *pssm_align='res'*, *biopython=False*)

Class from which Residue features are computed

Parameters

- **pdb** (*str, optional*) – path to pdb file. Defaults to None.
- **pssm** (*str, optional*) – path to pssm file. Defaults to None.
- **contact_distance** (*float, optional*) – cutoff distance for external edges. Defaults to 8.5.
- **internal_contact_distance** (*int, optional*) – cutoff distance for internal edges. Defaults to 3.
- **pssm_align** (*str, optional*) – [description]. Defaults to 'res'.

check_execs()

Checks if all the required external execs are installed.

Raises

OSError – if execs not found

get_graph(*db*)

Gets the interface graph nodes and edges given the internal and external distance threshold

Parameters

db (*[type]*) – sqldb database

get_node_features(*db*)

Assigns features to each node

get_edge_features()

Assigns distance feature to each edge

get_internal_edges(*db*)

Gets internal edges and the associated distances

onehot(*idx*, *size*)

One hot encoder

9.3 Graph Generator

9.4 Parrallel Graph Generator

class deeprank_gnn.GraphGenMP.**GraphHDF5**(*pdb_path*, *ref_path=None*, *graph_type='residue'*, *pssm_path=None*, *select=None*, *outfile='graph.hdf5'*, *nproc=1*, *use_tqdm=True*, *tmpdir='./'*, *limit=None*, *biopython=False*)

Master class from which graphs are computed :param pdb_path: path to the docking models :type pdb_path: str :param ref_path: path to the reference model. Defaults to None. :type ref_path: str, optional :param graph_type: Defaults to 'residue'. :type graph_type: str, optional :param pssm_path: path to the pssm file. Defaults to None. :type pssm_path: [type], optional :param select: filter files that starts with 'input'. Defaults to None. :type select: str, optional :param outfile: Defaults to 'graph.hdf5'. :type outfile: str, optional :param nproc: number of processors. Default to 1. :type nproc: int, optional :param use_tqdm: Default to True. :type use_tqdm: bool, optional :param tmpdir: Default to *./*. :type tmpdir: str, optional :param limit: Default to None. :type limit: int, optional :param >>> pdb_path = './data/pdb/1ATN/': :param >>> pssm_path = './data/pssm/1ATN/': :param >>> ref = './data/ref/1ATN/': :param >>> GraphHDF5(pdb_path=pdb_path: graph_type='residue', outfile='1AK4_residue.hdf5') :param ref_path=ref: graph_type='residue', outfile='1AK4_residue.hdf5') :param pssm_path=pssm_path: graph_type='residue', outfile='1AK4_residue.hdf5') :param : graph_type='residue', outfile='1AK4_residue.hdf5')

get_all_graphs(*pdbs*, *pssm*, *ref*, *outfile*, *use_tqdm=True*, *biopython=False*)

CHAPTER

TEN

NEURAL NETWORK TRAINING EVALUATION AND TEST

10.1 Neural Network

class deeprank_gnn.NeuralNet.**NeuralNet**(*database*, *Net*, *node_feature=['type', 'polarity', 'bsa']*, *edge_feature=['dist']*, *target='irmsd'*, *lr=0.01*, *batch_size=32*, *percent=[1.0, 0.0]*, *database_eval=None*, *index=None*, *class_weights=None*, *task=None*, *classes=[0, 1]*, *threshold=None*, *pretrained_model=None*, *shuffle=True*, *outdir='./'*, *cluster_nodes='mcl'*, *transform_sigmoid=False*)

Class from which the network is trained, evaluated and tested

Parameters

- **database** (*str, required*) – path(s) to hdf5 dataset(s). Unique hdf5 file or list of hdf5 files.
- **Net** (*function, required*) – neural network function (ex. GINet, Foutnet etc.)
- **node_feature** (*list, optional*) – type, charge, polarity, bsa (buried surface area), pssm, cons (pssm conservation information), ic (pssm information content), depth, hse (half sphere exposure). Defaults to ['type', 'polarity', 'bsa'].
- **edge_feature** (*list, optional*) – dist (distance). Defaults to ['dist'].
- **target** (*str, optional*) – irmsd, lrmsd, fnat, capri_class, bin_class, dockQ. Defaults to 'irmsd'.
- **lr** (*float, optional*) – learning rate. Defaults to 0.01.
- **batch_size** (*int, optional*) – defaults to 32.
- **percent** (*list, optional*) – divides the input dataset into a training and an evaluation set. Defaults to [1.0, 0.0].
- **database_eval** (*[type], optional*) – independent evaluation set. Defaults to None.
- **index** (*[type], optional*) – index of the molecules to consider. Defaults to None.
- **class_weights** (*[list or bool], optional*) – weights provided to the cross entropy loss function. The user can either input a list of weights or let DeepRanl-GNN (True) define weights based on the dataset content. Defaults to None.
- **task** (*str, optional*) – 'reg' for regression or 'class' for classification . Defaults to None.
- **classes** (*list, optional*) – define the dataset target classes in classification mode. Defaults to [0, 1].

- **threshold** (*int, optional*) – threshold to compute binary classification metrics. Defaults to 4.0.
- **pretrained_model** (*str, optional*) – path to pre-trained model. Defaults to None.
- **shuffle** (*bool, optional*) – shuffle the training set. Defaults to True.
- **outdir** (*str, optional*) – output directory. Defaults to ./
- **cluster_nodes** (*bool, optional*) – perform node clustering ('mcl' or 'louvain' algorithm). Default to 'mcl'.

load_pretrained_model(*database*, *Net*)

Loads pretrained model

Parameters

- **database** (*str*) – path to hdf5 file(s)
- **Net** (*function*) – neural network

load_model(*database*, *Net*, *database_eval*)

Loads model

Parameters

- **database** (*str*) – path to the hdf5 file(s) of the training set
- **Net** (*function*) – neural network
- **database_eval** (*str*) – path to the hdf5 file(s) of the evaluation set

Raises

ValueError – Invalid node clustering method.

put_model_to_device(*dataset*, *Net*)

Puts the model on the available device

Parameters

- **dataset** (*str*) – path to the hdf5 file(s)
- **Net** (*function*) – neural network

Raises

ValueError – Incorrect output shape

set_loss()

Sets the loss function (MSE loss for regression/ CrossEntropy loss for classification).

train(*nepoch=1*, *validate=False*, *save_model='last'*, *hdf5='train_data.hdf5'*, *save_epoch='intermediate'*, *save_every=5*)

Trains the model

Parameters

- **nepoch** (*int, optional*) – number of epochs. Defaults to 1.
- **validate** (*bool, optional*) – perform validation. Defaults to False.
- **save_model** (*last, best, optional*) – save the model. Defaults to 'last'
- **hdf5** (*str, optional*) – hdf5 output file
- **save_epoch** (*all, intermediate, optional*) –

- **save_every** (*int, optional*) – save data every n epoch if save_epoch == 'intermediate'. Defaults to 5

test(*database_test=None*, *threshold=4*, *hdf5='test_data.hdf5'*)

Tests the model

Parameters

- **database_test** (*[type], optional*) – test database
- **threshold** (*int, optional*) – threshold use to tranform data into binary values. Defaults to 4.
- **hdf5** (*str, optional*) – output hdf5 file. Defaults to 'test_data.hdf5'.

eval(*loader*)

Evaluates the model

Parameters
loader (*DataLoader*) – [description]

Return type
(tuple)

get_metrics(*data='eval'*, *threshold=4.0*, *binary=True*)

Computes the metrics needed

Parameters

- **data** (*str, optional*) – 'eval', 'train' or 'test'. Defaults to 'eval'.
- **threshold** (*float, optional*) – threshold use to tranform data into binary values. Defaults to 4.0.
- **binary** (*bool, optional*) – Transform data into binary data. Defaults to True.

compute_class_weights()

static print_epoch_data(*stage*, *epoch*, *loss*, *acc*, *time*)

Prints the data of each epoch

Parameters

- **stage** (*str*) – tain or valid
- **epoch** (*int*) – epoch number
- **loss** (*float*) – loss during that epoch
- **acc** (*float or None*) – accuracy
- **time** (*float*) – timing of the epoch

format_output(*pred*, *target=None*)

Format the network output depending on the task (classification/regression).

static update_name(*hdf5*, *outdir*)

Checks if the file already exists, if so, update the name

Parameters

- **hdf5** (*str*) – hdf5 file
- **outdir** (*str*) – output directory

Returns
update hdf5 name

Return type
str

`plot_loss`(*name=''*)

Plots the loss of the model as a function of the epoch

Parameters
name (*str, optional*) – name of the output file. Defaults to ‘’.

`plot_acc`(*name=''*)

Plots the accuracy of the model as a function of the epoch

Parameters
name (*str, optional*) – name of the output file. Defaults to ‘’.

`plot_hit_rate`(*data='eval'*, *threshold=4*, *mode='percentage'*, *name=''*)

Plots the hitrate as a function of the models’ rank

Parameters

- **data** (*str, optional*) – which stage to consider train/eval/test. Defaults to ‘eval’.
- **threshold** (*int, optional*) – defines the value to split into a hit (1) or a non-hit (0). Defaults to 4.
- **mode** (*str, optional*) – displays the hitrate as a number of hits (‘count’) or as a percentage (‘percantage’) . Defaults to ‘percentage’.

`plot_scatter`()

Scatters plot of the results.

`save_model`(*filename='model.pth.tar'*)

Saves the model to a file

Parameters
filename (*str, optional*) – name of the file. Defaults to ‘model.pth.tar’.

`load_params`(*filename*)

Loads the parameters of a pretrained model

Parameters
filename (*[type]*) – [description]

Returns
[description]

Return type
[type]

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

R

S

T

U