

---

# CogDL Documentation

**KEG**

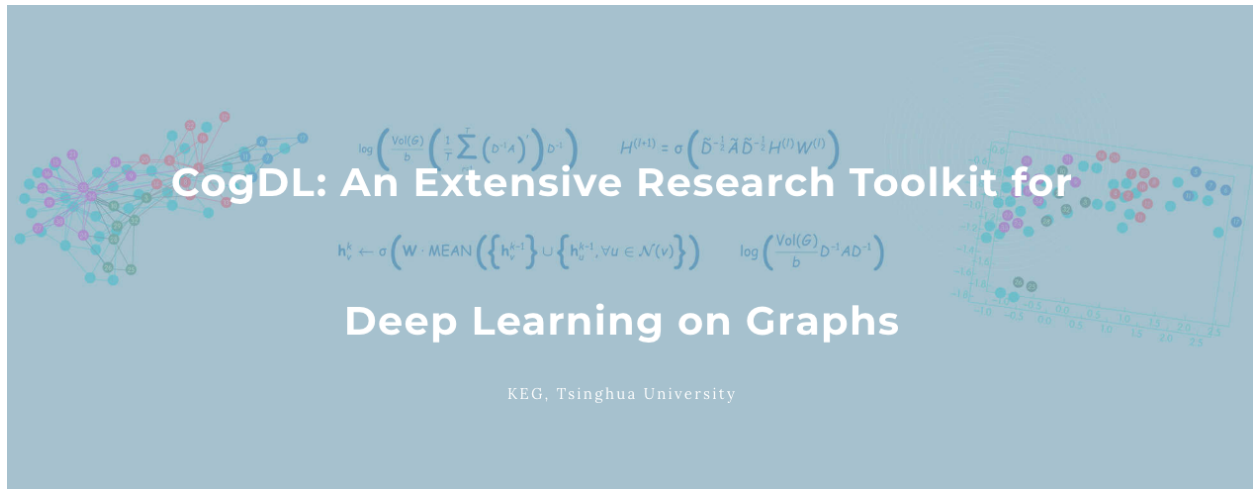
**Mar 03, 2021**



## CONTENTS:

<b>1</b>	<b>Install</b>	<b>3</b>
<b>2</b>	<b>Tutorial</b>	<b>5</b>
2.1	Create a model . . . . .	5
2.2	Create a dataset . . . . .	6
2.3	Create a task . . . . .	6
2.4	Combine model, dataset and task . . . . .	7
<b>3</b>	<b>Tasks</b>	<b>9</b>
3.1	Node Classification . . . . .	9
3.2	Unsupervised Node Classification . . . . .	11
3.3	Supervised Graph Classification . . . . .	14
3.4	Unsupervised Graph Classification . . . . .	16
<b>4</b>	<b>License</b>	<b>19</b>
<b>5</b>	<b>Citing</b>	<b>21</b>
<b>6</b>	<b>API Reference</b>	<b>23</b>
6.1	options . . . . .	23
6.2	utils . . . . .	24
6.3	layers . . . . .	24
6.4	data . . . . .	30
6.5	tasks . . . . .	41
6.6	datasets . . . . .	47
6.7	models . . . . .	56
<b>7</b>	<b>Indices and tables</b>	<b>91</b>
	<b>Python Module Index</b>	<b>93</b>
	<b>Index</b>	<b>95</b>





CogDL is a graph representation learning toolkit that allows researchers and developers to easily train and compare baseline or custom models for node classification, link prediction and other tasks on graphs. It provides implementations of many popular models, including: non-GNN Baselines like Deepwalk, LINE, NetMF, GNN Baselines like GCN, GAT, GraphSAGE.

CogDL provides these features:

- Task-Oriented: CogDL focuses on tasks on graphs and provides corresponding models, datasets, and leaderboards.
- Easy-Running: CogDL supports running multiple experiments simultaneously on multiple models and datasets under a specific task using multiple GPUs.
- Multiple Tasks: CogDL supports node classification and link prediction tasks on homogeneous/heterogeneous networks, as well as graph classification.
- Extensibility: You can easily add new datasets, models and tasks and conduct experiments for them!
- Supported tasks:
  - Node classification
  - Link prediction
  - Graph classification
  - Community detection (testing)
  - Social influence prediction (testing)
  - Graph reasoning (todo)
  - Graph pre-training (todo)
  - Combinatorial optimization on graphs (todo)



## INSTALL

- PyTorch version  $\geq$  1.0.0
- Python version  $\geq$  3.6
- PyTorch Geometric (optional)

Please follow the instructions here to install PyTorch: <https://github.com/pytorch/pytorch#installation>.

Please follow the instructions here to install PyTorch Geometric: [https://github.com/rusty1s/pytorch\\_geometric/#installation](https://github.com/rusty1s/pytorch_geometric/#installation).

Install other dependencies:

```
>>> pip install -e .
```





## TUTORIAL

This guide can help you start working with CogDL.

### 2.1 Create a model

Here, we will create a spectral clustering model, which is a very simple graph embedding algorithm. We name it `spectral.py` and put it in `cogdl/models/emb` directory.

First we import necessary library like `numpy`, `scipy`, `networkx`, `sklearn`, we also import API like `BaseModel` and `register_model` from `cogdl/models/` to build our new model:

```
import numpy as np
import networkx as nx
import scipy.sparse as sp
from sklearn import preprocessing
from .. import BaseModel, register_model
```

Then we use function decorator to declare new model for CogDL

```
@register_model('spectral')
class Spectral(BaseModel):
    (...)
```

We have to implement method `build_model_from_args` in `spectral.py`. If it need more parameters to train, we can use `add_args` to add model-specific arguments.

```
@staticmethod
def add_args(parser):
    """Add model-specific arguments to the parser."""
    pass

@classmethod
def build_model_from_args(cls, args):
    return cls(args.hidden_size)

def __init__(self, dimension):
    super(Spectral, self).__init__()
    self.dimension = dimension
```

Each new model should provide a `train` method to obtain representation.

```

def train(self, G):
    matrix = nx.normalized_laplacian_matrix(G).todense()
    matrix = np.eye(matrix.shape[0]) - np.asarray(matrix)
    ut, s, _ = sp.linalg.svds(matrix, self.dimension)
    emb_matrix = ut * np.sqrt(s)
    emb_matrix = preprocessing.normalize(emb_matrix, "l2")
    return emb_matrix

```

## 2.2 Create a dataset

In order to add a dataset into CogDL, you should know your dataset's format. We have provided several graph format like edgelist, matlab\_matrix and pyg. If your dataset is same as the 'ppi' dataset, which contains two matrices: 'network' and 'group', you can register your dataset directly use above code.

```

@register_dataset("ppi")
class PPIDataset(MatlabMatrix):
    def __init__(self):
        dataset, filename = "ppi", "Homo_sapiens"
        url = "http://snap.stanford.edu/node2vec/"
        path = osp.join(osp.dirname(osp.realpath(__file__)), "../..", "data", dataset)
        super(PPIDataset, self).__init__(path, filename, url)

```

You should declare the name of the dataset, the name of file and the url, where our script can download resource.

## 2.3 Create a task

In order to evaluate some methods on several datasets, we can build a task and evaluate learned representation. The BaseTask class are:

```

class BaseTask(object):
    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""
        pass

    def __init__(self, args):
        pass

    def train(self, num_epoch):
        raise NotImplementedError

```

we can create a subclass to implement 'train' method like CommunityDetection, which get representation of each node and apply clustering algorithm(K-means) to evaluate.

```

@register_task("community_detection")
class CommunityDetection(BaseTask):
    """Community Detection task."""

    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""
        parser.add_argument("--hidden-size", type=int, default=128)

```

(continues on next page)

(continued from previous page)

```

parser.add_argument("--num-shuffle", type=int, default=5)

def __init__(self, args):
    super(CommunityDetection, self).__init__(args)
    dataset = build_dataset(args)
    self.data = dataset[0]

    self.num_nodes, self.num_classes = self.data.y.shape
    self.label = np.argmax(self.data.y, axis=1)
    self.model = build_model(args)
    self.hidden_size = args.hidden_size
    self.num_shuffle = args.num_shuffle

def train(self):
    G = nx.Graph()
    G.add_edges_from(self.data.edge_index.t().tolist())
    embeddings = self.model.train(G)

    clusters = [30, 50, 70]
    all_results = defaultdict(list)
    for num_cluster in clusters:
        for _ in range(self.num_shuffle):
            model = KMeans(n_clusters=num_cluster).fit(embeddings)
            nmi_score = normalized_mutual_info_score(self.label, model.labels_)
            all_results[num_cluster].append(nmi_score)

    return dict(
        (
            f"normalized_mutual_info_score {num_cluster}",
            sum(all_results[num_cluster]) / len(all_results[num_cluster]),
        )
        for num_cluster in sorted(all_results.keys())
    )

```

## 2.4 Combine model, dataset and task

After create your model, dataset and task, we could combine them together to learn representation from a model on a dataset and evaluate its performance according to a task. We use 'build\_model', 'build\_dataset', 'build\_task' method to build them with coresponding parameters.

```

from cogdl.tasks import build_task
from cogdl.datasets import build_dataset
from cogdl.models import build_model
from cogdl.utils import build_args_from_dict

def test_deepwalk_ppi():
    default_dict = {'hidden_size': 64, 'num_shuffle': 1, 'cpu': True}
    args = build_args_from_dict(default_dict)

    # model, dataset and task parameters
    args.model = 'spectral'
    args.dataset = 'ppi'
    args.task = 'community_detection'

```

(continues on next page)

(continued from previous page)

```
# build model, dataset and task
dataset = build_dataset(args)
model = build_model(args)
task = build_task(args)

# train model and get evaluate results
ret = task.train()
print(ret)
```

## 3.1 Node Classification

In this tutorial, we will introduce a important task, node classification. In this task, we train a GNN model with partial node labels and use accuracy to measure the performance.

First we define the *NodeClassification* class.

```
@register_task("node_classification")
class NodeClassification(BaseTask):
    """Node classification task."""

    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""

    def __init__(self, args):
        super(NodeClassification, self).__init__(args)
```

Then we can build dataset according to args.

```
self.device = torch.device('cpu' if args.cpu else 'cuda')
dataset = build_dataset(args)
self.data = dataset.data
self.data.apply(lambda x: x.to(self.device))
args.num_features = dataset.num_features
args.num_classes = dataset.num_classes
```

After that, we can build model and use *Adam* to optimize the model.

```
model = build_model(args)
self.model = model.to(self.device)
self.patience = args.patience
self.max_epoch = args.max_epoch
self.optimizer = torch.optim.Adam(
    self.model.parameters(), lr=args.lr, weight_decay=args.weight_decay
)
```

We provide a training loop for node classification task. For each epoch, we first call *\_train\_step* to optimize our model and then call *\_test\_step* to compute the accuracy and loss.

```
def train(self):
    epoch_iter = tqdm(range(self.max_epoch))
    patience = 0
```

(continues on next page)

```

best_score = 0
best_loss = np.inf
max_score = 0
min_loss = np.inf
for epoch in epoch_iter:
    self._train_step()
    train_acc, _ = self._test_step(split="train")
    val_acc, val_loss = self._test_step(split="val")
    epoch_iter.set_description(
        f"Epoch: {epoch:03d}, Train: {train_acc:.4f}, Val: {val_acc:.4f}"
    )
    if val_loss <= min_loss or val_acc >= max_score:
        if val_loss <= best_loss: # and val_acc >= best_score:
            best_loss = val_loss
            best_score = val_acc
            best_model = copy.deepcopy(self.model)
            min_loss = np.min((min_loss, val_loss))
            max_score = np.max((max_score, val_acc))
            patience = 0
        else:
            patience += 1
            if patience == self.patience:
                self.model = best_model
                epoch_iter.close()
                break

def _train_step(self):
    self.model.train()
    self.optimizer.zero_grad()
    self.model.loss(self.data).backward()
    self.optimizer.step()

def _test_step(self, split="val"):
    self.model.eval()
    logits = self.model.predict(self.data)
    _, mask = list(self.data(f"{split}_mask"))[0]
    loss = F.nll_loss(logits[mask], self.data.y[mask])

    pred = logits[mask].max(1)[1]
    acc = pred.eq(self.data.y[mask]).sum().item() / mask.sum().item()
    return acc, loss

```

Finally, we compute the accuracy scores of test set for the trained model.

```

test_acc, _ = self._test_step(split="test")
print(f"Test accuracy = {test_acc}")
return dict(Acc=test_acc)

```

The overall implementation of *NodeClassification* is at ([https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/node\\_classification.py](https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/node_classification.py)).

To run *NodeClassification*, we can use the following command:

```

python scripts/train.py --task node_classification --dataset cora citeseer --model_
↳ pyg_gcn pyg_gat --seed 0 1 --max-epoch 500

```

Then We get experimental results like this:

Variant	Acc
('cora', 'pyg_gcn')	0.7785±0.0165
('cora', 'pyg_gat')	0.7925±0.0045
('citeseer', 'pyg_gcn')	0.6535±0.0195
('citeseer', 'pyg_gat')	0.6675±0.0025

## 3.2 Unsupervised Node Classification

In this tutorial, we will introduce a important task, unsupervised node classification. In this task, we usually apply L2 normalized logistic regression to train a classifier and use F1-score to measure the performance.

First we define the *UnsupervisedNodeClassification* class, which has two parameters *hidden-size* and *num-shuffle*. *hidden-size* represents the dimension of node representation, while *num-shuffle* means the shuffle times in classifier.

```
@register_task("unsupervised_node_classification")
class UnsupervisedNodeClassification(BaseTask):
    """Node classification task."""

    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""
        # fmt: off
        parser.add_argument("--hidden-size", type=int, default=128)
        parser.add_argument("--num-shuffle", type=int, default=5)
        # fmt: on

    def __init__(self, args):
        super(UnsupervisedNodeClassification, self).__init__(args)
```

Then we can build dataset according to input graph's type, and get *self.label\_matrix*.

```
dataset = build_dataset(args)
self.data = dataset[0]
if issubclass(dataset.__class__.__bases__[0], InMemoryDataset):
    self.num_nodes = self.data.y.shape[0]
    self.num_classes = dataset.num_classes
    self.label_matrix = np.zeros((self.num_nodes, self.num_classes), dtype=int)
    self.label_matrix[range(self.num_nodes), self.data.y] = 1
    self.data.edge_attr = self.data.edge_attr.t()
else:
    self.label_matrix = self.data.y
    self.num_nodes, self.num_classes = self.data.y.shape
```

After that, we can build model and run *model.train(G)* to obtain node representation.

```
self.model = build_model(args)
self.model_name = args.model
self.hidden_size = args.hidden_size
self.num_shuffle = args.num_shuffle
self.save_dir = args.save_dir
self.enhance = args.enhance
self.args = args
self.is_weighted = self.data.edge_attr is not None
```

(continues on next page)

(continued from previous page)

```

def train(self):
    G = nx.Graph()
    if self.is_weighted:
        edges, weight = (
            self.data.edge_index.t().tolist(),
            self.data.edge_attr.tolist(),
        )
        G.add_weighted_edges_from(
            [(edges[i][0], edges[i][1], weight[0][i]) for i in range(len(edges))]
        )
    else:
        G.add_edges_from(self.data.edge_index.t().tolist())
    embeddings = self.model.train(G)

```

The spectral propagation in ProNE can improve the quality of representation learned from other methods, so we can use `enhance_emb` to enhance performance.

```

if self.enhance is True:
    embeddings = self.enhance_emb(G, embeddings)

def enhance_emb(self, G, embs):
    A = sp.csr_matrix(nx.adjacency_matrix(G))
    self.args.model = 'prone'
    self.args.step, self.args.theta, self.args.mu = 5, 0.5, 0.2
    model = build_model(self.args)
    embs = model._chebyshev_gaussian(A, embs)
    return embs

```

When the embeddings are obtained, we can save them at `self.save_dir`.

```

# Map node2id
features_matrix = np.zeros((self.num_nodes, self.hidden_size))
for vid, node in enumerate(G.nodes()):
    features_matrix[node] = embeddings[vid]

self.save_emb(features_matrix)

def save_emb(self, embs):
    name = os.path.join(self.save_dir, self.model_name + '_emb.npy')
    np.save(name, embs)

```

At last, we evaluate embedding via run `num_shuffle` times classification under different training ratio with `features_matrix` and `label_matrix`.

```

return self._evaluate(features_matrix, label_matrix, self.num_shuffle)

def _evaluate(self, features_matrix, label_matrix, num_shuffle):
    # shuffle, to create train/test groups
    shuffles = []
    for _ in range(num_shuffle):
        shuffles.append(skshuffle(features_matrix, label_matrix))

    # score each train/test group
    all_results = defaultdict(list)
    training_percents = [0.1, 0.3, 0.5, 0.7, 0.9]

```

(continues on next page)



(continued from previous page)

```

for train_percent in training_percents:
    for shuf in shuffles:

```

In each shuffle, split data into two parts(training and testing) and use *LogisticRegression* to evaluate.

```

X, y = shuf

training_size = int(train_percent * self.num_nodes)

X_train = X[:training_size, :]
y_train = y[:training_size, :]

X_test = X[training_size:, :]
y_test = y[training_size:, :]

clf = TopKRanker(LogisticRegression())
clf.fit(X_train, y_train)

# find out how many labels should be predicted
top_k_list = list(map(int, y_test.sum(axis=1).T.tolist()[0]))
preds = clf.predict(X_test, top_k_list)
result = f1_score(y_test, preds, average="micro")
all_results[train_percent].append(result)

```

Node in graph may have multiple labels, so we conduct multilabel classification built from TopKRanker.

```

from sklearn.multiclass import OneVsRestClassifier

class TopKRanker(OneVsRestClassifier):
    def predict(self, X, top_k_list):
        assert X.shape[0] == len(top_k_list)
        probs = np.asarray(super(TopKRanker, self).predict_proba(X))
        all_labels = sp.lil_matrix(probs.shape)

        for i, k in enumerate(top_k_list):
            probs_ = probs[i, :]
            labels = self.classes_[probs_.argsort()[-k:]].tolist()
            for label in labels:
                all_labels[i, label] = 1
        return all_labels

```

Finally, we get the results of Micro-F1 score under different training ratio for different models on datasets.

```

return dict(
    (
        f"Micro-F1 {train_percent}",
        sum(all_results[train_percent]) / len(all_results[train_percent]),
    )
    for train_percent in sorted(all_results.keys())
)

```

The overall implementation of *UnsupervisedNodeClassification* is at ([https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised\\_node\\_classification.py](https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised_node_classification.py)).

To run *UnsupervisedNodeClassification*, we can use following instruction:

```
python scripts/train.py --task unsupervised_node_classification --dataset ppi_
↳wikipedia --model deepwalk prone -seed 0 1
```

Then We get experimental results like this:

Variant	Micro-F1 0.1	Micro-F1 0.3	Micro-F1 0.5	Micro-F1 0.7	Micro-F1 0.9
('ppi', 'deepwalk')	0.1547±0.0002	0.1846±0.0002	0.2033±0.0015	0.2161±0.0009	0.2243±0.0018
('ppi', 'prone')	0.1777±0.0016	0.2214±0.0020	0.2397±0.0015	0.2486±0.0022	0.2607±0.0096
('wikipedia', 'deepwalk')	0.4255±0.0027	0.4712±0.0005	0.4916±0.0011	0.5011±0.0017	0.5166±0.0043
('wikipedia', 'prone')	0.4834±0.0009	0.5320±0.0020	0.5504±0.0045	0.5586±0.0022	0.5686±0.0072

### 3.3 Supervised Graph Classification

In this section, we will introduce the implementation “Graph classification task”.

#### Task Design

- Set up “SupervisedGraphClassification” class, which has two specific parameters.
  - degree-feature*: Use one-hot node degree as node feature, for datasets such as Imdb-binary and Imdb-multi, which don't have node features.
  - gamma*: Multiplicative factor of learning rate decay.
  - lr*: Learning rate.
- Build dataset convert it to a list of *Data* defined in Cogdl. Specially, we reformat the data according to the input format of specific models. *generate\_data* is implemented to convert dataset.

```
dataset = build_dataset(args)
self.data = self.generate_data(dataset, args)

def generate_data(self, dataset, args):
    if "ModelNet" in str(type(dataset).__name__):
        train_set, test_set = dataset.get_all()
        args.num_features = 3
        return {"train": train_set, "test": test_set}
    else:
        datalist = []
        if isinstance(dataset[0], Data):
            return dataset
        for idata in dataset:
            data = Data()
            for key in idata.keys:
                data[key] = idata[key]
            datalist.append(data)

        if args.degree_feature:
            datalist = node_degree_as_feature(datalist)
            args.num_features = datalist[0].num_features
        return datalist
...
```

- Then we build model and can run *train* to train the model.

```

def train(self):
    for epoch in epoch_iter:
        self._train_step()
        val_acc, val_loss = self._test_step(split="valid")
        # ...
        return dict(Acc=test_acc)

def _train_step(self):
    self.model.train()
    loss_n = 0
    for batch in self.train_loader:
        batch = batch.to(self.device)
        self.optimizer.zero_grad()
        output, loss = self.model(batch)
        loss_n += loss.item()
        loss.backward()
        self.optimizer.step()

def _test_step(self, split):
    """split in ['train', 'test', 'valid']"""
    # ...
    return acc, loss

```

The overall implementation of GraphClassification is at ([https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/graph\\_classification.py](https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/graph_classification.py)).

### Create a model

To create a model for task graph classification, the following functions have to be implemented.

1. *add\_args(parser)*: add necessary hyper-parameters used in model.

```

@staticmethod
def add_args(parser):
    parser.add_argument("--hidden-size", type=int, default=128)
    parser.add_argument("--num-layers", type=int, default=2)
    parser.add_argument("--lr", type=float, default=0.001)
    # ...

```

2. *build\_model\_from\_args(cls, args)*: this function is called in 'task' to build model.
3. *split\_dataset(cls, dataset, args)*: split train/validation/test data and return correspondent dataloader according to requirement of model.

```

def split_dataset(cls, dataset, args):
    random.shuffle(dataset)
    train_size = int(len(dataset) * args.train_ratio)
    test_size = int(len(dataset) * args.test_ratio)
    bs = args.batch_size
    train_loader = DataLoader(dataset[:train_size], batch_size=bs)
    test_loader = DataLoader(dataset[-test_size:], batch_size=bs)
    if args.train_ratio + args.test_ratio < 1:
        valid_loader = DataLoader(dataset[train_size:-test_size], batch_size=bs)
    else:
        valid_loader = test_loader
    return train_loader, valid_loader, test_loader

```

4. *forward*: forward propagation, and the return should be (predication, loss) or (prediction, None), respectively for training and test. Input parameters of *forward* is class *Batch*, which

```

def forward(self, batch):
    h = batch.x
    layer_rep = [h]
    for i in range(self.num_layers-1):
        h = self.gin_layers[i](h, batch.edge_index)
        h = self.batch_norm[i](h)
        h = F.relu(h)
        layer_rep.append(h)

    final_score = 0
    for i in range(self.num_layers):
        pooled = scatter_add(layer_rep[i], batch.batch, dim=0)
        final_score += self.dropout(self.linear_prediction[i](pooled))
    final_score = F.softmax(final_score, dim=-1)
    if batch.y is not None:
        loss = self.loss(final_score, batch.y)
        return final_score, loss
    return final_score, None

```

### Run

To run GraphClassification, we can use the following command:

```
python scripts/train.py --task graph_classification --dataset proteins --model gin_
↳diffpool sortpool dgcnn --seed 0 1
```

Then We get experimental results like this:

Variants	Acc
('proteins', 'gin')	0.7286±0.0598
('proteins', 'diffpool')	0.7530±0.0589
('proteins', 'sortpool')	0.7411±0.0269
('proteins', 'dgcnn')	0.6677±0.0355
('proteins', 'patchy_san')	0.7550±0.0812

## 3.4 Unsupervised Graph Classification

In this section, we will introduce the implementation “Unsupervised graph classification task”.

### Task Design

1. Set up “UnsupervisedGraphClassification” class, which has two specific parameters.
  - *num-shuffle* : Shuffle times in classifier
  - *degree-feature*: Use one-hot node degree as node feature, for datasets such as Imdb-binary and Imdb-multi, which don't have node features.
  - *lr*: learning

```

@register_task("unsupervised_graph_classification")
class UnsupervisedGraphClassification(BaseTask):
    r"""Unsupervised graph classification"""
    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""

```

(continues on next page)

(continued from previous page)

```

    # fmt: off
    parser.add_argument("--num-shuffle", type=int, default=10)
    parser.add_argument("--degree-feature", dest="degree_feature", action="store_
↪true")
    parser.add_argument("--lr", type=float, default=0.001)
    # fmt: on
    def __init__(self, args):
        # ...

```

## 2. Build dataset and convert it to a list of *Data* defined in Cogdl.

```

dataset = build_dataset(args)
self.label = np.array([data.y for data in dataset])
self.data = [
    Data(x=data.x, y=data.y, edge_index=data.edge_index, edge_attr=data.edge_attr,
        pos=data.pos).apply(lambda x:x.to(self.device))
    for data in dataset
]

```

## 3. Then we build model and can run *train* to train the model and obtain graph representation. In this part, the training process of shallow models and deep models are implemented separately.

```

self.model = build_model(args)
self.model = self.model.to(self.device)

def train(self):
    if self.use_nn:
        # deep neural network models
        epoch_iter = tqdm(range(self.epoch))
        for epoch in epoch_iter:
            loss_n = 0
            for batch in self.data_loader:
                batch = batch.to(self.device)
                predict, loss = self.model(batch.x, batch.edge_index, batch.batch)
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                loss_n += loss.item()
            # ...
    else:
        # shallow models
        prediction, loss = self.model(self.data)
        label = self.label

```

## 4. When graph representation is obtained, we evaluate the embedding with *SVM* via running *num\_shuffle* times under different training ratio. You can also call *save\_emb* to save the embedding.

```

return self._evaluate(prediction, label)
def _evaluate(self, embedding, labels):
    # ...
    for training_percent in training_percent:
        for shuf in shuffles:
            # ...
            clf = SVC()
            clf.fit(X_train, y_train)
            preds = clf.predict(X_test)

```

(continues on next page)

```
... # ...
```

The overall implementation of `UnsupervisedGraphClassification` is at ([https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised\\_graph\\_classification.py](https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised_graph_classification.py)).

### Create a model

To create a model for task unsupervised graph classification, the following functions have to be implemented.

1. `add_args(parser)`: add necessary hyper-parameters used in model.

```
@staticmethod
def add_args(parser):
    parser.add_argument("--hidden-size", type=int, default=128)
    parser.add_argument("--nn", type=bool, default=False)
    parser.add_argument("--lr", type=float, default=0.001)
    # ...
```

2. `build_model_from_args(cls, args)`: this function is called in 'task' to build model.
3. `forward`: For shallow models, this function runs as training process of model and will be called only once; For deep neural network models, this function is actually the forward propagation process and will be called many times.

```
# shallow model
def forward(self, graphs):
    # ...
    self.model = Doc2Vec(
        self.doc_collections,
        ...
    )
    vectors = np.array([self.model["g_"+str(i)] for i in range(len(graphs))])
    return vectors, None
```

### Run

To run `UnsupervisedGraphClassification`, we can use the following command:

```
python scripts/train.py --task unsupervised_graph_classification --dataset proteins --
↪model dgk graph2vec
```

Then we get experimental results like this:

Variant	Acc
('proteins', 'dgk')	0.7259±0.0118
('proteins', 'graph2vec')	0.7330±0.0043
('proteins', 'infograph')	0.7393±0.0070

**LICENSE**

MIT License

Copyright (c) 2020

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.





## CITING

- [Perozzi et al. (2014): Deepwalk: Online learning of social representations](<http://arxiv.org/abs/1403.6652>)
- [Tang et al. (2015): Line: Large-scale information network embedding](<http://arxiv.org/abs/1503.03578>)
- [Grover and Leskovec. (2016): node2vec: Scalable feature learning for networks](<http://dl.acm.org/citation.cfm?doid=2939672.2939754>)- [Cao et al. (2015):Grarep: Learning graph representations with global structural information ](<http://dl.acm.org/citation.cfm?doid=2806416.2806512>)
- [Ou et al. (2016): Asymmetric transitivity preserving graph embedding](<http://dl.acm.org/citation.cfm?doid=2939672.2939751>)
- [Qiu et al. (2017): Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec](<http://arxiv.org/abs/1710.02971>)
- [Qiu et al. (2019): NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization](<http://arxiv.org/abs/1710.02971>)
- [Zhang et al. (2019): Spectral Network Embedding: A Fast and Scalable Method via Sparsity](<http://arxiv.org/abs/1806.02623>)
- [Kipf and Welling (2016): Semi-Supervised Classification with Graph Convolutional Networks](<https://arxiv.org/abs/1609.02907>)
- [Hamilton et al. (2017): Inductive Representation Learning on Large Graphs](<https://arxiv.org/abs/1706.02216>)
- [Veličković et al. (2017): Graph Attention Networks](<https://arxiv.org/abs/1710.10903>)
- [Ding et al. (2018): Semi-supervised Learning on Graphs with Generative Adversarial Nets](<https://arxiv.org/abs/1809.00130>)
- [Han et al. (2019): GroupRep: Unsupervised Structural Representation Learning for Groups in Networks](<https://www.overleaf.com/read/nqxjtkmmgmff>)
- [Zhang et al. (2019): Revisiting Graph Convolutional Networks: Neighborhood Aggregation and Network Sampling](<https://www.overleaf.com/read/xzykmvvhxjmxxy>)
- [Zhang et al. (2019): Co-training Graph Convolutional Networks with Network Redundancy](<https://www.overleaf.com/read/fbhqqgzqgmyn>)
- [Qiu et al. (2019): NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization](<http://keg.cs.tsinghua.edu.cn/jietang/publications/www19-Qiu-et-al-NetSMF-Large-Scale-Network-Embedding.pdf>)
- [Zhang et al. (2019): ProNE: Fast and Scalable Network Representation Learning](<https://www.overleaf.com/read/dhgpkyfhdhj>)
- [Cen et al. (2019): Representation Learning for Attributed Multiplex Heterogeneous Network](<https://arxiv.org/abs/1905.01669>)



## API REFERENCE

This page contains auto-generated API reference documentation<sup>1</sup>.

### 6.1 options

#### 6.1.1 Module Contents

##### Functions

---

*get\_parser()*

---

*add\_task\_args(parser)*

---

*add\_dataset\_args(parser)*

---

*add\_model\_args(parser)*

---

*get\_training\_parser()*

---

*get\_display\_data\_parser()*

---

*get\_download\_data\_parser()*

---

*parse\_args\_and\_arch(parser, args)*

The parser doesn't know about model-specific args, so we parse twice.

---

`options.get_parser()`

`options.add_task_args(parser)`

`options.add_dataset_args(parser)`

`options.add_model_args(parser)`

`options.get_training_parser()`

`options.get_display_data_parser()`

`options.get_download_data_parser()`

---

<sup>1</sup> Created with sphinx-autoapi

`options.parse_args_and_arch(parser, args)`

The parser doesn't know about model-specific args, so we parse twice.

## 6.2 utils

### 6.2.1 Module Contents

#### Classes

---

*ArgClass*

---

#### Functions

---

*build\_args\_from\_dict(dic)*

---

---

*add\_remaining\_self\_loops(edge\_index, edge\_weight, fill\_value, num\_nodes)*

---

**class** `utils.ArgClass`

Bases: `object`

`utils.build_args_from_dict(dic)`

`utils.add_remaining_self_loops(edge_index, edge_weight, fill_value, num_nodes)`

`utils.args`

## 6.3 layers

### 6.3.1 Submodules

`layers.gcc_module`

#### Module Contents

#### Classes

---

<i>SELayer</i>	Squeeze-and-excitation networks
<i>ApplyNodeFunc</i>	Update the node feature hv with MLP, BN and ReLU.
<i>MLP</i>	MLP with linear output
<i>UnsupervisedGAT</i>	
<i>UnsupervisedMPNN</i>	MPNN from
<i>UnsupervisedGIN</i>	GIN model
<i>GraphEncoder</i>	MPNN from

---

```

class layers.gcc_module.SELayer (in_channels, se_channels)
    Bases: torch.nn.Module

    Squeeze-and-excitation networks

    forward (self, x)

class layers.gcc_module.ApplyNodeFunc (mlp, use_selayer)
    Bases: torch.nn.Module

    Update the node feature hv with MLP, BN and ReLU.

    forward (self, h)

class layers.gcc_module.MLP (num_layers, input_dim, hidden_dim, output_dim, use_selayer)
    Bases: torch.nn.Module

    MLP with linear output

    forward (self, x)

class layers.gcc_module.UnsupervisedGAT (node_input_dim, node_hidden_dim,
                                         edge_input_dim, num_layers, num_heads)
    Bases: torch.nn.Module

    forward (self, g, n_feat, e_feat)

class layers.gcc_module.UnsupervisedMPNN (output_dim=32, node_input_dim=32,
                                           node_hidden_dim=32,
                                           edge_input_dim=32, edge_hidden_dim=32,
                                           num_step_message_passing=6,
                                           lstm_as_gate=False)

    Bases: torch.nn.Module

    MPNN from Neural Message Passing for Quantum Chemistry

    node_input_dim [int] Dimension of input node feature, default to be 15.
    edge_input_dim [int] Dimension of input edge feature, default to be 15.
    output_dim [int] Dimension of prediction, default to be 12.
    node_hidden_dim [int] Dimension of node feature in hidden layers, default to be 64.
    edge_hidden_dim [int] Dimension of edge feature in hidden layers, default to be 128.
    num_step_message_passing [int] Number of message passing steps, default to be 6.
    num_step_set2set [int] Number of set2set steps
    num_layer_set2set [int] Number of set2set layers

    forward (self, g, n_feat, e_feat)
        Predict molecule labels

        g [DGLGraph] Input DGLGraph for molecule(s)

        n_feat [tensor of dtype float32 and shape (B1, D1)] Node features. B1 for number of nodes and D1 for
            the node feature size.

        e_feat [tensor of dtype float32 and shape (B2, D2)] Edge features. B2 for number of edges and D2 for the
            edge feature size.

        res : Predicted labels

```

```
class layers.gcc_module.UnsupervisedGIN(num_layers, num_mlp_layers, input_dim, hidden_dim, output_dim, final_dropout, learn_eps, graph_pooling_type, neighbor_pooling_type, use_slayer)
```

Bases: torch.nn.Module

GIN model

**forward** (*self*, *g*, *h*, *efeat*)

```
class layers.gcc_module.GraphEncoder(positional_embedding_size=32, max_node_freq=8, max_edge_freq=8, max_degree=128, freq_embedding_size=32, degree_embedding_size=32, output_dim=32, node_hidden_dim=32, edge_hidden_dim=32, num_layers=6, num_heads=4, num_step_set2set=6, num_layer_set2set=3, norm=False, gnn_model='mpnn', degree_input=False, lstm_as_gate=False)
```

Bases: torch.nn.Module

MPNN from [Neural Message Passing for Quantum Chemistry](#)

**node\_input\_dim** [int] Dimension of input node feature, default to be 15.

**edge\_input\_dim** [int] Dimension of input edge feature, default to be 15.

**output\_dim** [int] Dimension of prediction, default to be 12.

**node\_hidden\_dim** [int] Dimension of node feature in hidden layers, default to be 64.

**edge\_hidden\_dim** [int] Dimension of edge feature in hidden layers, default to be 128.

**num\_step\_message\_passing** [int] Number of message passing steps, default to be 6.

**num\_step\_set2set** [int] Number of set2set steps

**num\_layer\_set2set** [int] Number of set2set layers

**forward** (*self*, *g*, *return\_all\_outputs=False*)

Predict molecule labels

**g** [DGLGraph] Input DGLGraph for molecule(s)

**n\_feat** [tensor of dtype float32 and shape (B1, D1)] Node features. B1 for number of nodes and D1 for the node feature size.

**e\_feat** [tensor of dtype float32 and shape (B2, D2)] Edge features. B2 for number of edges and D2 for the edge feature size.

res : Predicted labels

**layers.maggregator**

## Module Contents

### Classes

---

*MeanAggregator*

---

---

```

class layers.maggregator.MeanAggregator (in_channels, out_channels, improved=False,
                                         cached=False, bias=True)
    Bases: torch.nn.Module
    static norm (x, edge_index)
    forward (self, x, edge_index, edge_weight=None, bias=True)
    update (self, aggr_out)
    __repr__ (self)

```

`layers.mixhop_layer`

## Module Contents

### Classes

---

*MixHopLayer*

---

```

class layers.mixhop_layer.MixHopLayer (num_features, adj_pows, dim_per_pow)
    Bases: torch.nn.Module
    reset_parameters (self)
    adj_pow_x (self, x, adj, p)
    forward (self, x, edge_index)
layers.mixhop_layer.layer

```

`layers.se_layer`

## Module Contents

### Classes

---

<i>SELayer</i>	Squeeze-and-excitation networks
----------------	---------------------------------

---

```

class layers.se_layer.SELayer (in_channels, se_channels)
    Bases: torch.nn.Module
    Squeeze-and-excitation networks
    forward (self, x)

```

`layers.srgcn_module`

### Module Contents

#### Classes

---

*NodeAttention*

---

*EdgeAttention*

---

*Identity*

---

*Gaussian*

---

*PPR*

---

*HeatKernel*

---

*NormIdentity*

---

*RowUniform*

---

*RowSoftmax*

---

*ColumnUniform*

---

*SymmetryNorm*

---

#### Functions

---

*act\_attention*(attn\_type)

---

*act\_normalization*(norm\_type)

---

*act\_map*(act)

---

**class** `layers.srgcn_module.NodeAttention` (*in\_feat*)

Bases: `torch.nn.Module`

**forward** (*self*, *x*, *edge\_index*, *edge\_attr*)

**class** `layers.srgcn_module.EdgeAttention` (*in\_feat*)

Bases: `torch.nn.Module`

**forward** (*self*, *x*, *edge\_index*, *edge\_attr*)

**class** `layers.srgcn_module.Identity` (*in\_feat*)

Bases: `torch.nn.Module`

**forward** (*self*, *x*, *edge\_index*, *edge\_attr*)



```

class layers.srgcn_module.Gaussian (in_feat)
    Bases: torch.nn.Module

    forward (self, x, edge_index, edge_attr)

class layers.srgcn_module.PPR (in_feat)
    Bases: torch.nn.Module

    forward (self, x, edge_index, edge_attr)

class layers.srgcn_module.HeatKernel (in_feat)
    Bases: torch.nn.Module

    forward (self, x, edge_index, edge_attr)

layers.srgcn_module.act_attention (attn_type)

class layers.srgcn_module.NormIdentity
    Bases: torch.nn.Module

    forward (self, edge_index, edge_attr, N)

class layers.srgcn_module.RowUniform
    Bases: torch.nn.Module

    forward (self, edge_index, edge_attr, N)

class layers.srgcn_module.RowSoftmax
    Bases: torch.nn.Module

    forward (self, edge_index, edge_attr, N)

class layers.srgcn_module.ColumnUniform
    Bases: torch.nn.Module

    forward (self, edge_index, edge_attr, N)

class layers.srgcn_module.SymmetryNorm
    Bases: torch.nn.Module

    forward (self, edge_index, edge_attr, N)

layers.srgcn_module.act_normalization (norm_type)
layers.srgcn_module.act_map (act)

```

## 6.3.2 Package Contents

### Classes

---

*MeanAggregator*

---

*SELayer*

---

Squeeze-and-excitation networks

---

*MixHopLayer*

---

```

class layers.MeanAggregator (in_channels, out_channels, improved=False, cached=False,
                             bias=True)
    Bases: torch.nn.Module

    static norm (x, edge_index)

```

**forward** (*self*, *x*, *edge\_index*, *edge\_weight=None*, *bias=True*)

**update** (*self*, *aggr\_out*)

**\_\_repr\_\_** (*self*)

**class** layers.**SELayer** (*in\_channels*, *se\_channels*)

Bases: torch.nn.Module

Squeeze-and-excitation networks

**forward** (*self*, *x*)

**class** layers.**MixHopLayer** (*num\_features*, *adj\_pows*, *dim\_per\_pow*)

Bases: torch.nn.Module

**reset\_parameters** (*self*)

**adj\_pow\_x** (*self*, *x*, *adj*, *p*)

**forward** (*self*, *x*, *edge\_index*)

## 6.4 data

### 6.4.1 Submodules

data.batch

#### Module Contents

#### Classes

---

*Batch*

A plain old python object modeling a batch of graphs as one big

---

**class** data.batch.**Batch** (*batch=None*, *\*\*kwargs*)

Bases: cogdl.data.Data

A plain old python object modeling a batch of graphs as one big (dicconnected) graph. With `cogdl.data.Data` being the base class, all its methods can also be used here. In addition, single graphs can be reconstructed via the assignment vector `batch`, which maps each node to its respective graph identifier.

**static from\_data\_list** (*data\_list*, *follow\_batch=[]*)

Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector `batch` is created on the fly. Additionally, creates assignment batch vectors for each key in `follow_batch`.

**cumsum** (*self*, *key*, *item*)

If `True`, the attribute `key` with content `item` should be added up cumulatively before concatenated together.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

**to\_data\_list** (*self*)

Reconstructs the list of `torch_geometric.data.Data` objects from the batch object. The batch object must have been created via `from_data_list()` in order to be able reconstruct the initial objects.

**property num\_graphs** (*self*)  
Returns the number of graphs in the batch.

`data.data`

## Module Contents

### Classes

---

<i>Data</i>	A plain old python object modeling a single graph with various
-------------	--

---

**class** `data.data.Data` (*x=None, edge\_index=None, edge\_attr=None, y=None, pos=None*)

Bases: `object`

A plain old python object modeling a single graph with various (optional) attributes:

#### Args:

**x (Tensor, optional): Node feature matrix with shape :obj:`[num\_nodes, num\_node\_features]`.** (default: `None`)

**edge\_index (LongTensor, optional): Graph connectivity in COO format with shape `[2, num_edges]`.** (default: `None`)

**edge\_attr (Tensor, optional): Edge feature matrix with shape `[num_edges, num_edge_features]`.** (default: `None`)

**y (Tensor, optional): Graph or node targets with arbitrary shape.** (default: `None`)

**pos (Tensor, optional): Node position matrix with shape `[num_nodes, num_dimensions]`.** (default: `None`)

The data object is not restricted to these attributes and can be extended by any other additional data.

**static from\_dict** (*dictionary*)

Creates a data object from a python dictionary.

**\_\_getitem\_\_** (*self, key*)

Gets the data of the attribute `key`.

**\_\_setitem\_\_** (*self, key, value*)

Sets the attribute `key` to `value`.

**property keys** (*self*)

Returns all names of graph attributes.

**\_\_len\_\_** (*self*)

Returns the number of all present attributes.

**\_\_contains\_\_** (*self, key*)

Returns `True`, if the attribute `key` is present in the data.

**\_\_iter\_\_** (*self*)

Iterates over all present attributes in the data, yielding their attribute names and content.

`__call__` (*self*, \**keys*)

Iterates over all attributes \**keys* in the data, yielding their attribute names and content. If \**keys* is not given this method will iterative over all present attributes.

`cat_dim` (*self*, *key*, *value*)

Returns the dimension in which the attribute *key* with content *value* gets concatenated when creating batches.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

`__inc__` (*self*, *key*, *value*)

“Returns the incremental count to cumulatively increase the value of the next attribute of *key* when creating batches.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

`property num_edges` (*self*)

Returns the number of edges in the graph.

`property num_features` (*self*)

Returns the number of features per node in the graph.

`property num_nodes` (*self*)

`is_coalesced` (*self*)

Returns `True`, if edge indices are ordered and do not contain duplicate entries.

`apply` (*self*, *func*, \**keys*)

Applies the function *func* to all attributes \**keys*. If \**keys* is not given, *func* is applied to all present attributes.

`contiguous` (*self*, \**keys*)

Ensures a contiguous memory layout for all attributes \**keys*. If \**keys* is not given, all present attributes are ensured to have a contiguous memory layout.

`to` (*self*, *device*, \**keys*)

Performs tensor dtype and/or device conversion to all attributes \**keys*. If \**keys* is not given, the conversion is applied to all present attributes.

`cuda` (*self*, \**keys*)

`clone` (*self*)

`__repr__` (*self*)

Return `repr(self)`.

`data.dataloader`

## Module Contents

### Classes

---

<code>DataLoader</code>	Data loader which merges data objects from a
<code>DataListLoader</code>	Data loader which merges data objects from a
<code>DenseDataLoader</code>	Data loader which merges data objects from a

---

**class** `data.dataloader.DataLoader` (*dataset, batch\_size=1, shuffle=True, \*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

**Args:** `dataset` (Dataset): The dataset from which to load the data. `batch_size` (int, optional): How may samples per batch to load.

(default: 1)

**shuffle** (bool, optional): If set to **True**, the data will be reshuffled at every epoch (default: **True**)

**class** `data.dataloader.DataListLoader` (*dataset, batch\_size=1, shuffle=True, \*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a python list.

---

**Note:** This data loader should be used for multi-gpu support via `cogdl.nn.DataParallel`.

---

**Args:** `dataset` (Dataset): The dataset from which to load the data. `batch_size` (int, optional): How may samples per batch to load.

(default: 1)

**shuffle** (bool, optional): If set to **True**, the data will be reshuffled at every epoch (default: **True**)

**class** `data.dataloader.DenseDataLoader` (*dataset, batch\_size=1, shuffle=True, \*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

---

**Note:** To make use of this data loader, all graphs in the dataset needs to have the same shape for each its attributes. Therefore, this data loader should only be used when working with *dense* adjacency matrices.

---

**Args:** `dataset` (Dataset): The dataset from which to load the data. `batch_size` (int, optional): How may samples per batch to load.

(default: 1)

**shuffle** (bool, optional): If set to **True**, the data will be reshuffled at every epoch (default: **True**)

`data.dataset`

### Module Contents

#### Classes

---

`Dataset`

Dataset base class for creating graph datasets.

---

#### Functions

---

`to_list(x)`

---

`files_exist(files)`

---

`data.dataset.to_list(x)`

`data.dataset.files_exist(files)`

**class** `data.dataset.Dataset` (*root, transform=None, pre\_transform=None, pre\_filter=None*)

Bases: `torch.utils.data.Dataset`

Dataset base class for creating graph datasets. See [here](#) for the accompanying tutorial.

**Args:** *root* (string): Root directory where the dataset should be saved. *transform* (callable, optional): A function/transform that takes in an

`cogdl.data.Data` object and returns a transformed version. The data object will be transformed before every access. (default: `None`)

**pre\_transform** (callable, optional): A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk. (default: `None`)

**pre\_filter** (callable, optional): A function that takes in an `cogdl.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset. (default: `None`)

**property** `raw_file_names` (*self*)

The name of the files to find in the `self.raw_dir` folder in order to skip the download.

**property** `processed_file_names` (*self*)

The name of the files to find in the `self.processed_dir` folder in order to skip the processing.

**abstract** `download` (*self*)

Downloads the dataset to the `self.raw_dir` folder.

**abstract** `process` (*self*)

Processes the dataset to the `self.processed_dir` folder.

**abstract** `__len__` (*self*)

The number of examples in the dataset.

**abstract** `get` (*self, idx*)

Gets the data object at index `idx`.

**property num\_features** (*self*)

Returns the number of features per node in the graph.

**property raw\_paths** (*self*)

The filepaths to find in order to skip the download.

**property processed\_paths** (*self*)

The filepaths to find in the `self.processed_dir` folder in order to skip the processing.

**\_\_download** (*self*)

**\_\_process** (*self*)

**\_\_getitem\_\_** (*self*, *idx*)

Gets the data object at index `idx` and transforms it (in case a `self.transform` is given).

**\_\_repr\_\_** (*self*)

**data.download**

## Module Contents

### Functions

---

`download_url(url, folder, name=None, log=True)` Downloads the content of an URL to a specific folder.

---

`data.download.download_url(url, folder, name=None, log=True)`

Downloads the content of an URL to a specific folder.

**Args:** `url` (string): The url. `folder` (string): The folder. `log` (bool, optional): If `False`, will not print anything to the

console. (default: `True`)

**data.extract**

## Module Contents

### Functions

---

`maybe_log(path, log=True)`

---

`extract_tar(path, folder, mode='r:gz', log=True)` Extracts a tar archive to a specific folder.

`extract_zip(path, folder, log=True)` Extracts a zip archive to a specific folder.

`extract_bz2(path, folder, log=True)`

---

`extract_gz(path, folder, log=True)`

---

`data.extract.maybe_log(path, log=True)`

`data.extract.extract_tar(path, folder, mode='r:gz', log=True)`

Extracts a tar archive to a specific folder.

**Args:** path (string): The path to the tar archive. folder (string): The folder. mode (string, optional): The compression mode. (default: "r:gz") log (bool, optional): If `False`, will not print anything to the console. (default: `True`)

`data.extract.extract_zip` (*path, folder, log=True*)  
 Extracts a zip archive to a specific folder.

**Args:** path (string): The path to the tar archive. folder (string): The folder. log (bool, optional): If `False`, will not print anything to the console. (default: `True`)

`data.extract.extract_bz2` (*path, folder, log=True*)

`data.extract.extract_gz` (*path, folder, log=True*)

**data.makedirs**

**Module Contents**

**Functions**

---

*makedirs*(path)

---

`data.makedirs.makedirs` (*path*)

**6.4.2 Package Contents**

**Classes**

<i>Data</i>	A plain old python object modeling a single graph with various
<i>Batch</i>	A plain old python object modeling a batch of graphs as one big
<i>Dataset</i>	Dataset base class for creating graph datasets.
<i>DataLoader</i>	Data loader which merges data objects from a
<i>DataListLoader</i>	Data loader which merges data objects from a
<i>DenseDataLoader</i>	Data loader which merges data objects from a

**Functions**

<i>download_url</i> (url, folder, name=None, log=True)	Downloads the content of an URL to a specific folder.
<i>extract_tar</i> (path, folder, mode='r:gz', log=True)	Extracts a tar archive to a specific folder.
<i>extract_zip</i> (path, folder, log=True)	Extracts a zip archive to a specific folder.
<i>extract_bz2</i> (path, folder, log=True)	
<i>extract_gz</i> (path, folder, log=True)	

---



**class** `data.Data` ( $x=None$ ,  $edge\_index=None$ ,  $edge\_attr=None$ ,  $y=None$ ,  $pos=None$ )

Bases: `object`

A plain old python object modeling a single graph with various (optional) attributes:

**Args:**

**x (Tensor, optional): Node feature matrix with shape `:obj: [num_nodes, num_node_features]`.** (default: `None`)

**edge\_index (LongTensor, optional): Graph connectivity in COO format with shape `[2, num_edges]`.** (default: `None`)

**edge\_attr (Tensor, optional): Edge feature matrix with shape `[num_edges, num_edge_features]`.** (default: `None`)

**y (Tensor, optional): Graph or node targets with arbitrary shape.** (default: `None`)

**pos (Tensor, optional): Node position matrix with shape `[num_nodes, num_dimensions]`.** (default: `None`)

The data object is not restricted to these attributes and can be extended by any other additional data.

**static from\_dict** (*dictionary*)

Creates a data object from a python dictionary.

**\_\_getitem\_\_** (*self*, *key*)

Gets the data of the attribute *key*.

**\_\_setitem\_\_** (*self*, *key*, *value*)

Sets the attribute *key* to *value*.

**property keys** (*self*)

Returns all names of graph attributes.

**\_\_len\_\_** (*self*)

Returns the number of all present attributes.

**\_\_contains\_\_** (*self*, *key*)

Returns `True`, if the attribute *key* is present in the data.

**\_\_iter\_\_** (*self*)

Iterates over all present attributes in the data, yielding their attribute names and content.

**\_\_call\_\_** (*self*, *\*keys*)

Iterates over all attributes *\*keys* in the data, yielding their attribute names and content. If *\*keys* is not given this method will iterative over all present attributes.

**cat\_dim** (*self*, *key*, *value*)

Returns the dimension in which the attribute *key* with content *value* gets concatenated when creating batches.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

**\_\_inc\_\_** (*self*, *key*, *value*)

“Returns the incremental count to cumulatively increase the value of the next attribute of *key* when creating batches.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

**property num\_edges** (*self*)

Returns the number of edges in the graph.

**property num\_features** (*self*)

Returns the number of features per node in the graph.

**property num\_nodes** (*self*)

**is\_coalesced** (*self*)

Returns `True`, if edge indices are ordered and do not contain duplicate entries.

**apply** (*self*, *func*, *\*keys*)

Applies the function *func* to all attributes *\*keys*. If *\*keys* is not given, *func* is applied to all present attributes.

**contiguous** (*self*, *\*keys*)

Ensures a contiguous memory layout for all attributes *\*keys*. If *\*keys* is not given, all present attributes are ensured to have a contiguous memory layout.

**to** (*self*, *device*, *\*keys*)

Performs tensor dtype and/or device conversion to all attributes *\*keys*. If *\*keys* is not given, the conversion is applied to all present attributes.

**cuda** (*self*, *\*keys*)

**clone** (*self*)

**\_\_repr\_\_** (*self*)

Return `repr(self)`.

**class** `data.Batch` (*batch=None*, *\*\*kwargs*)

Bases: `cogdl.data.Data`

A plain old python object modeling a batch of graphs as one big (dicconnected) graph. With `cogdl.data.Data` being the base class, all its methods can also be used here. In addition, single graphs can be reconstructed via the assignment vector *batch*, which maps each node to its respective graph identifier.

**static from\_data\_list** (*data\_list*, *follow\_batch=[]*)

Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector *batch* is created on the fly. Additionally, creates assignment batch vectors for each key in *follow\_batch*.

**cumsum** (*self*, *key*, *item*)

If `True`, the attribute *key* with content *item* should be added up cumulatively before concatenated together.

---

**Note:** This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

**to\_data\_list** (*self*)

Reconstructs the list of `torch_geometric.data.Data` objects from the batch object. The batch object must have been created via `from_data_list()` in order to be able reconstruct the initial objects.

**property num\_graphs** (*self*)

Returns the number of graphs in the batch.

**class** `data.Dataset` (*root*, *transform=None*, *pre\_transform=None*, *pre\_filter=None*)

Bases: `torch.utils.data.Dataset`

Dataset base class for creating graph datasets. See [here](#) for the accompanying tutorial.

**Args:** *root* (string): Root directory where the dataset should be saved. *transform* (callable, optional): A function/transform that takes in an

`cogdl.data.Data` object and returns a transformed version. The data object will be transformed before every access. (default: `None`)

**pre\_transform** (callable, optional): A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk. (default: `None`)

**pre\_filter** (callable, optional): A function that takes in an `cogdl.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset. (default: `None`)

**property** `raw_file_names` (*self*)

The name of the files to find in the `self.raw_dir` folder in order to skip the download.

**property** `processed_file_names` (*self*)

The name of the files to find in the `self.processed_dir` folder in order to skip the processing.

**abstract** `download` (*self*)

Downloads the dataset to the `self.raw_dir` folder.

**abstract** `process` (*self*)

Processes the dataset to the `self.processed_dir` folder.

**abstract** `__len__` (*self*)

The number of examples in the dataset.

**abstract** `get` (*self*, *idx*)

Gets the data object at index *idx*.

**property** `num_features` (*self*)

Returns the number of features per node in the graph.

**property** `raw_paths` (*self*)

The filepaths to find in order to skip the download.

**property** `processed_paths` (*self*)

The filepaths to find in the `self.processed_dir` folder in order to skip the processing.

`_download` (*self*)

`_process` (*self*)

`__getitem__` (*self*, *idx*)

Gets the data object at index *idx* and transforms it (in case a `self.transform` is given).

`__repr__` (*self*)

**class** `data.DataLoader` (*dataset*, *batch\_size=1*, *shuffle=True*, *\*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

**Args:** *dataset* (Dataset): The dataset from which to load the data. *batch\_size* (int, optional): How many samples per batch to load.

(default: 1)

**shuffle (bool, optional):** If set to **True**, the data will be reshuffled at every epoch (default: `True`)

**class** `data.DataListLoader` (*dataset, batch\_size=1, shuffle=True, \*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a python list.

---

**Note:** This data loader should be used for multi-gpu support via `cogdl.nn.DataParallel`.

---

**Args:** `dataset` (Dataset): The dataset from which to load the data. `batch_size` (int, optional): How many samples per batch to load.

(default: 1)

**shuffle (bool, optional):** If set to **True**, the data will be reshuffled at every epoch (default: `True`)

**class** `data.DenseDataLoader` (*dataset, batch\_size=1, shuffle=True, \*\*kwargs*)

Bases: `torch.utils.data.DataLoader`

Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

---

**Note:** To make use of this data loader, all graphs in the dataset needs to have the same shape for each its attributes. Therefore, this data loader should only be used when working with *dense* adjacency matrices.

---

**Args:** `dataset` (Dataset): The dataset from which to load the data. `batch_size` (int, optional): How many samples per batch to load.

(default: 1)

**shuffle (bool, optional):** If set to **True**, the data will be reshuffled at every epoch (default: `True`)

`data.download_url` (*url, folder, name=None, log=True*)

Downloads the content of an URL to a specific folder.

**Args:** `url` (string): The url. `folder` (string): The folder. `log` (bool, optional): If `False`, will not print anything to the

console. (default: `True`)

`data.extract_tar` (*path, folder, mode='r:gz', log=True*)

Extracts a tar archive to a specific folder.

**Args:** `path` (string): The path to the tar archive. `folder` (string): The folder. `mode` (string, optional): The compression mode. (default: `"r:gz"`) `log` (bool, optional): If `False`, will not print anything to the

console. (default: `True`)

`data.extract_zip` (*path, folder, log=True*)

Extracts a zip archive to a specific folder.

**Args:** `path` (string): The path to the tar archive. `folder` (string): The folder. `log` (bool, optional): If `False`, will not print anything to the

console. (default: `True`)

`data.extract_bz2` (*path, folder, log=True*)

`data.extract_gz` (*path, folder, log=True*)

## 6.5 tasks

### 6.5.1 Submodules

`tasks.base_task`

#### Module Contents

#### Classes

---

*BaseTask*

---

```
class tasks.base_task.BaseTask (args)
    Bases: object

    static add_args (parser)
        Add task-specific arguments to the parser.

    abstract train (self, num_epoch)
```

`tasks.graph_classification`

#### Module Contents

#### Classes

---

<i>GraphClassification</i>	Supervised graph classification task.
----------------------------	---------------------------------------

---

#### Functions

---

<i>node_degree_as_feature</i> ( <i>data</i> )	Set each node feature as one-hot encoding of degree
<i>uniform_node_feature</i> ( <i>data</i> )	Set each node feature to the same

---

```
tasks.graph_classification.node_degree_as_feature (data)
    Set each node feature as one-hot encoding of degree :param data: a list of class Data :return: a list of class Data
```

```
tasks.graph_classification.uniform_node_feature (data)
    Set each node feature to the same
```

```
class tasks.graph_classification.GraphClassification (args)
    Bases: tasks.BaseTask

    Supervised graph classification task.

    static add_args (parser)
        Add task-specific arguments to the parser.

    train (self)
```

```
_train (self)
_train_step (self)
_test_step (self, split='val')
_kfold_train (self)
generate_data (self, dataset, args)
```

`tasks.heterogeneous_node_classification`

### Module Contents

#### Classes

---

<i>HeterogeneousNodeClassification</i>	Heterogeneous Node classification task.
--	---

---

```
class tasks.heterogeneous_node_classification.HeterogeneousNodeClassification (args)
    Bases: tasks.BaseTask
    Heterogeneous Node classification task.
    static add_args (parser)
        Add task-specific arguments to the parser.
    train (self)
    _train_step (self)
    _test_step (self, split='val')
```

`tasks.link_prediction`

### Module Contents

#### Classes

---

<i>LinkPrediction</i>
-----------------------

---

#### Functions

---

<i>divide_data</i> (input_list, division_rate)
<i>randomly_choose_false_edges</i> (nodes, true_edges, num)
<i>gen_node_pairs</i> (train_data, test_data, negative_ratio=5)
<i>get_score</i> (embs, node1, node2)

---

continues on next page

Table 26 – continued from previous page

---

`evaluate(embs, true_edges, false_edges)`


---

`tasks.link_prediction.divide_data(input_list, division_rate)`
`tasks.link_prediction.randomly_choose_false_edges(nodes, true_edges, num)`
`tasks.link_prediction.gen_node_pairs(train_data, test_data, negative_ratio=5)`
`tasks.link_prediction.get_score(embs, node1, node2)`
`tasks.link_prediction.evaluate(embs, true_edges, false_edges)`
**class** `tasks.link_prediction.LinkPrediction(args)`

 Bases: `tasks.BaseTask`
**static add\_args** (`parser`)

Add task-specific arguments to the parser.

**train** (`self`)

**tasks.multiplex\_link\_prediction**

## Module Contents

### Classes

---

`MultiplexLinkPrediction`


---

### Functions

---

`get_score(embs, node1, node2)`


---

`evaluate(embs, true_edges, false_edges)`


---

`tasks.multiplex_link_prediction.get_score(embs, node1, node2)`
`tasks.multiplex_link_prediction.evaluate(embs, true_edges, false_edges)`
**class** `tasks.multiplex_link_prediction.MultiplexLinkPrediction(args)`

 Bases: `tasks.BaseTask`
**static add\_args** (`parser`)

Add task-specific arguments to the parser.

**train** (`self`)

`tasks.multiplex_node_classification`

### Module Contents

#### Classes

---

<code><i>MultiplexNodeClassification</i></code>	Node classification task.
---	---------------------------

---

**class** `tasks.multiplex_node_classification.MultiplexNodeClassification` (*args*)

Bases: `tasks.BaseTask`

Node classification task.

**static add\_args** (*parser*)  
Add task-specific arguments to the parser.

**train** (*self*)

`tasks.node_classification`

### Module Contents

#### Classes

---

<code><i>NodeClassification</i></code>	Node classification task.
--	---------------------------

---

**class** `tasks.node_classification.NodeClassification` (*args*, *dataset=None*,  
*model=None*)

Bases: `tasks.BaseTask`

Node classification task.

**static add\_args** (*parser*)  
Add task-specific arguments to the parser.

**train** (*self*)

**\_train\_step** (*self*)

**\_test\_step** (*self*, *split='val'*)

`tasks.node_classification_sampling`

### Module Contents

#### Classes

---

<code><i>NodeClassificationSampling</i></code>	Node classification task with sampling.
--	---

---



## Functions

---

```
get_batches(train_nodes,          train_labels,
             batch_size=64, shuffle=True)
```

---

```
tasks.node_classification_sampling.get_batches (train_nodes,          train_labels,
                                                batch_size=64, shuffle=True)
```

```
class tasks.node_classification_sampling.NodeClassificationSampling (args)
```

Bases: *tasks.BaseTask*

Node classification task with sampling.

**static add\_args** (*parser*)

Add task-specific arguments to the parser.

**train** (*self*)

**\_train\_step** (*self*)

**\_test\_step** (*self, split='val'*)

**tasks.unsupervised\_graph\_classification**

## Module Contents

### Classes

---

<i>UnsupervisedGraphClassification</i>	Unsupervised graph classification
--	-----------------------------------

---

```
class tasks.unsupervised_graph_classification.UnsupervisedGraphClassification (args)
```

Bases: *tasks.BaseTask*

Unsupervised graph classification

**static add\_args** (*parser*)

Add task-specific arguments to the parser.

**train** (*self*)

**save\_emb** (*self, embs*)

**\_evaluate** (*self, embeddings, labels*)

**tasks.unsupervised\_node\_classification**

## Module Contents

### Classes

---

<i>UnsupervisedNodeClassification</i>	Node classification task.
<i>TopKRanker</i>	

---

```
tasks.unsupervised_node_classification.pyg = False
```

```
class tasks.unsupervised_node_classification.UnsupervisedNodeClassification(args)  
    Bases: tasks.BaseTask
```

Node classification task.

```
static add_args (parser)  
    Add task-specific arguments to the parser.
```

```
enhance_emb (self, G, embs)
```

```
save_emb (self, embs)
```

```
train (self)
```

```
_evaluate (self, features_matrix, label_matrix, num_shuffle)
```

```
class tasks.unsupervised_node_classification.TopKRanker  
    Bases: sklearn.multiclass.OneVsRestClassifier
```

```
predict (self, X, top_k_list)
```

## 6.5.2 Package Contents

### Classes

---

*BaseTask*

---

### Functions

---

<i>register_task</i> ( <i>name</i> )	New task types can be added to cogdl with the <i>register_task()</i>
--------------------------------------	--

---

<i>build_task</i> ( <i>args, dataset=None, model=None</i> )
---

---

```
class tasks.BaseTask(args)
```

Bases: *object*

```
static add_args (parser)  
    Add task-specific arguments to the parser.
```

```
abstract train (self, num_epoch)
```

```
tasks.TASK_REGISTRY
```

```
tasks.register_task(name)
```

New task types can be added to cogdl with the *register\_task()* function decorator.

For example:

```
@register_task('node_classification')  
class NodeClassification(BaseTask):  
    (...)
```

**Args:** *name* (str): the name of the task

`tasks.task_name`

`tasks.build_task` (*args*, *dataset=None*, *model=None*)

## 6.6 datasets

### 6.6.1 Submodules

`datasets.gatne`

#### Module Contents

#### Classes

<i>GatneDataset</i>	The network datasets “Amazon”, “Twitter” and “YouTube” from the
<i>AmazonDataset</i>	The network datasets “Amazon”, “Twitter” and “YouTube” from the
<i>TwitterDataset</i>	The network datasets “Amazon”, “Twitter” and “YouTube” from the
<i>YouTubeDataset</i>	The network datasets “Amazon”, “Twitter” and “YouTube” from the

#### Functions

*read\_gatne\_data*(*folder*)

`datasets.gatne.read_gatne_data` (*folder*)

**class** `datasets.gatne.GatneDataset` (*root*, *name*)

Bases: `cogdl.data.Dataset`

The network datasets “Amazon”, “Twitter” and “YouTube” from the “Representation Learning for Attributed Multiplex Heterogeneous Network” paper.

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset (“Amazon”, “Twitter”, “YouTube”).

**url** = <https://github.com/THUDM/GATNE/raw/master/data>

**property** `raw_file_names` (*self*)

**property** `processed_file_names` (*self*)

**get** (*self*, *idx*)

**download** (*self*)

**process** (*self*)

**\_\_repr\_\_** (*self*)

**class** `datasets.gatne.AmazonDataset`

Bases: `datasets.gatne.GatneDataset`

The network datasets “Amazon”, “Twitter” and “YouTube” from the “Representation Learning for Attributed Multiplex Heterogeneous Network” paper.

**Args:** `root` (string): Root directory where the dataset should be saved. `name` (string): The name of the dataset ("Amazon", "Twitter", "YouTube").

**class** `datasets.gatne.TwitterDataset`

Bases: `datasets.gatne.GatneDataset`

The network datasets “Amazon”, “Twitter” and “YouTube” from the “Representation Learning for Attributed Multiplex Heterogeneous Network” paper.

**Args:** `root` (string): Root directory where the dataset should be saved. `name` (string): The name of the dataset ("Amazon", "Twitter", "YouTube").

**class** `datasets.gatne.YouTubeDataset`

Bases: `datasets.gatne.GatneDataset`

The network datasets “Amazon”, “Twitter” and “YouTube” from the “Representation Learning for Attributed Multiplex Heterogeneous Network” paper.

**Args:** `root` (string): Root directory where the dataset should be saved. `name` (string): The name of the dataset ("Amazon", "Twitter", "YouTube").

`datasets.gcc_data`

### Module Contents

#### Classes

---

`Edgelist`

---

`USAAirportDataset`

---

**class** `datasets.gcc_data.Edgelist` (*root, name*)

Bases: `cogdl.data.Dataset`

`url = https://github.com/cenyk1230/gcc-data/raw/master`

**property** `raw_file_names` (*self*)

**property** `processed_file_names` (*self*)

**download** (*self*)

**get** (*self, idx*)

**process** (*self*)

**class** `datasets.gcc_data.USAAirportDataset`

Bases: `datasets.gcc_data.Edgelist`

## `datasets.gtn_data`

### Module Contents

#### Classes

<code>GTNDataset</code>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<code>ACM_GTNDataset</code>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<code>DBLP_GTNDataset</code>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<code>IMDB_GTNDataset</code>	The network datasets “ACM”, “DBLP” and “IMDB” from the

#### Functions

<code>untar(path, fname, deleteTar=True)</code>	Unpacks the given archive file to the same directory, then (by default)
---	---

`datasets.gtn_data.untar` (*path, fname, deleteTar=True*)

Unpacks the given archive file to the same directory, then (by default) deletes the archive file.

**class** `datasets.gtn_data.GTNDataset` (*root, name*)

Bases: `cogdl.data.Dataset`

The network datasets “ACM”, “DBLP” and “IMDB” from the “[Graph Transformer Networks](#)” paper.

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset (“gtn-acm”, “gtn-dblp”, “gtn-imdb”).

**property** `raw_file_names` (*self*)

**property** `processed_file_names` (*self*)

**read\_gtn\_data** (*self, folder*)

**get** (*self, idx*)

**apply\_to\_device** (*self, device*)

**download** (*self*)

**process** (*self*)

**\_\_repr\_\_** (*self*)

**class** `datasets.gtn_data.ACM_GTNDataset`

Bases: `datasets.gtn_data.GTNDataset`

The network datasets “ACM”, “DBLP” and “IMDB” from the “[Graph Transformer Networks](#)” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("gtn-acm", "gtn-dblp", "gtn-imdb").

**class** datasets.gtn\_data.DBLP\_GTNDataset

Bases: *datasets.gtn\_data.GTNDataset*

The network datasets “ACM”, “DBLP” and “IMDB” from the “Graph Transformer Networks” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("gtn-acm", "gtn-dblp", "gtn-imdb").

**class** datasets.gtn\_data.IMDB\_GTNDataset

Bases: *datasets.gtn\_data.GTNDataset*

The network datasets “ACM”, “DBLP” and “IMDB” from the “Graph Transformer Networks” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("gtn-acm", "gtn-dblp", "gtn-imdb").

## datasets.han\_data

### Module Contents

#### Classes

<i>HANDataset</i>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<i>ACM_HANDataset</i>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<i>DBLP_HANDataset</i>	The network datasets “ACM”, “DBLP” and “IMDB” from the
<i>IMDB_HANDataset</i>	The network datasets “ACM”, “DBLP” and “IMDB” from the

#### Functions

<i>untar</i> (path, fname, deleteTar=True)	Unpacks the given archive file to the same directory, then (by default)
<i>sample_mask</i> (idx, l)	Create mask.

datasets.han\_data.untar (path, fname, deleteTar=True)

Unpacks the given archive file to the same directory, then (by default) deletes the archive file.

datasets.han\_data.sample\_mask (idx, l)

Create mask.

**class** datasets.han\_data.HANDataset (root, name)

Bases: cogdl.data.Dataset

The network datasets “ACM”, “DBLP” and “IMDB” from the “Heterogeneous Graph Attention Network” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("han-acm",

"han-dblp", "han-imdb").

**property raw\_file\_names** (*self*)

**property processed\_file\_names** (*self*)

**read\_gtn\_data** (*self*, *folder*)

**get** (*self*, *idx*)

**apply\_to\_device** (*self*, *device*)

**download** (*self*)

**process** (*self*)

**\_\_repr\_\_** (*self*)

**class** datasets.han\_data.ACM\_HANDataset

Bases: *datasets.han\_data.HANDataset*

The network datasets “ACM”, “DBLP” and “IMDB” from the “Heterogeneous Graph Attention Network” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("han-acm",

"han-dblp", "han-imdb").

**class** datasets.han\_data.DBLP\_HANDataset

Bases: *datasets.han\_data.HANDataset*

The network datasets “ACM”, “DBLP” and “IMDB” from the “Heterogeneous Graph Attention Network” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("han-acm",

"han-dblp", "han-imdb").

**class** datasets.han\_data.IMDB\_HANDataset

Bases: *datasets.han\_data.HANDataset*

The network datasets “ACM”, “DBLP” and “IMDB” from the “Heterogeneous Graph Attention Network” paper.

**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("han-acm",

"han-dblp", "han-imdb").

**datasets.matlab\_matrix**

## Module Contents

### Classes

<i>MatlabMatrix</i>	networks from the <a href="http://leitang.net/code/social-dimension/data/">http://leitang.net/code/social-dimension/data/</a> or <a href="http://snap.stanford.edu/node2vec/">http://snap.stanford.edu/node2vec/</a>
<i>BlogcatalogDataset</i>	networks from the <a href="http://leitang.net/code/social-dimension/data/">http://leitang.net/code/social-dimension/data/</a> or <a href="http://snap.stanford.edu/node2vec/">http://snap.stanford.edu/node2vec/</a>
<i>FlickrDataset</i>	networks from the <a href="http://leitang.net/code/social-dimension/data/">http://leitang.net/code/social-dimension/data/</a> or <a href="http://snap.stanford.edu/node2vec/">http://snap.stanford.edu/node2vec/</a>
<i>WikipediaDataset</i>	networks from the <a href="http://leitang.net/code/social-dimension/data/">http://leitang.net/code/social-dimension/data/</a> or <a href="http://snap.stanford.edu/node2vec/">http://snap.stanford.edu/node2vec/</a>
<i>PPIDataset</i>	networks from the <a href="http://leitang.net/code/social-dimension/data/">http://leitang.net/code/social-dimension/data/</a> or <a href="http://snap.stanford.edu/node2vec/">http://snap.stanford.edu/node2vec/</a>

**class** `datasets.matlab_matrix.MatlabMatrix` (*root, name, url*)

Bases: `cogdl.data.Dataset`

networks from the <http://leitang.net/code/social-dimension/data/> or <http://snap.stanford.edu/node2vec/>

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset ("Blogcatalog").

**property** `raw_file_names` (*self*)

**property** `processed_file_names` (*self*)

**download** (*self*)

**get** (*self, idx*)

**process** (*self*)

**class** `datasets.matlab_matrix.BlogcatalogDataset`

Bases: `datasets.matlab_matrix.MatlabMatrix`

networks from the <http://leitang.net/code/social-dimension/data/> or <http://snap.stanford.edu/node2vec/>

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset ("Blogcatalog").

**class** `datasets.matlab_matrix.FlickrDataset`

Bases: `datasets.matlab_matrix.MatlabMatrix`

networks from the <http://leitang.net/code/social-dimension/data/> or <http://snap.stanford.edu/node2vec/>

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset ("Blogcatalog").

**class** `datasets.matlab_matrix.WikipediaDataset`

Bases: `datasets.matlab_matrix.MatlabMatrix`

networks from the <http://leitang.net/code/social-dimension/data/> or <http://snap.stanford.edu/node2vec/>

**Args:** *root* (string): Root directory where the dataset should be saved. *name* (string): The name of the dataset ("Blogcatalog").

**class** `datasets.matlab_matrix.PPIDataset`

Bases: `datasets.matlab_matrix.MatlabMatrix`

networks from the <http://leitang.net/code/social-dimension/data/> or <http://snap.stanford.edu/node2vec/>



**Args:** root (string): Root directory where the dataset should be saved. name (string): The name of the dataset ("Blogcatalog").

## `datasets.pyg`

### Module Contents

#### Classes

---

*CoraDataset*

---

*CiteSeerDataset*

---

*PubMedDataset*

---

*RedditDataset*

---

*MUTAGDataset*

---

*ImdbBinaryDataset*

---

*ImdbMultiDataset*

---

*CollabDataset*

---

*ProtainsDataset*

---

*RedditBinary*

---

*RedditMulti5K*

---

*RedditMulti12K*

---

*PTCMRDataset*

---

*NCT1Dataset*

---

*NCT109Dataset*

---

*ENZYMES*

---

*QM9Dataset*

---

```
class datasets.pyg.CoraDataset
    Bases: torch_geometric.datasets.Planetoid

class datasets.pyg.CiteSeerDataset
    Bases: torch_geometric.datasets.Planetoid

class datasets.pyg.PubMedDataset
    Bases: torch_geometric.datasets.Planetoid
```

```
class datasets.pyg.RedditDataset
    Bases: torch_geometric.datasets.Reddit

class datasets.pyg.MUTAGDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.ImdbBinaryDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.ImdbMultiDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.CollabDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.ProteinsDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.RedditBinary
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.RedditMulti5K
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.RedditMulti12K
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.PTCMRDataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.NCT1Dataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.NCT109Dataset
    Bases: torch_geometric.datasets.TUDataset

class datasets.pyg.ENZYMES
    Bases: torch_geometric.datasets.TUDataset
    __getitem__ (self, idx)

class datasets.pyg.QM9Dataset
    Bases: torch_geometric.datasets.QM9
```

`datasets.pyg_modelnet`

### Module Contents

#### Classes

---

*ModelNet10*

---

*ModelNet40*

---

*ModelNetData10*

---

*ModelNetData40*

---

```

class datasets.pyg_modelnet.ModelNet10 (train)
    Bases: torch_geometric.datasets.ModelNet

class datasets.pyg_modelnet.ModelNet40 (train)
    Bases: torch_geometric.datasets.ModelNet

class datasets.pyg_modelnet.ModelNetData10
    Bases: torch_geometric.datasets.ModelNet

    get_all (self)
    __getitem__ (self, item)
    __len__ (self)
    property train_index (self)
    property test_index (self)

class datasets.pyg_modelnet.ModelNetData40
    Bases: torch_geometric.datasets.ModelNet

    get_all (self)
    __getitem__ (self, item)
    __len__ (self)
    property train_index (self)
    property test_index (self)

```

## 6.6.2 Package Contents

### Functions

---

<code>register_dataset(name)</code>	New dataset types can be added to cogdl with the <code>register_dataset()</code>
<code>build_dataset(args)</code>	
<code>build_dataset_from_name(dataset)</code>	

---

```
datasets.pyg = False
```

```
datasets.dgl_import = False
```

```
datasets.DATASET_REGISTRY
```

```
datasets.register_dataset (name)
```

New dataset types can be added to cogdl with the `register_dataset()` function decorator.

For example:

```

@register_dataset ('my_dataset')
class MyDataset ():
    (...)

```

**Args:** name (str): the name of the dataset

`datasets.dataset_name`

`datasets.build_dataset (args)`

`datasets.build_dataset_from_name (dataset)`

## 6.7 models

### 6.7.1 Subpackages

`models.emb`

#### Submodules

`models.emb.deepwalk`

#### Module Contents

#### Classes

---

*DeepWalk*

The DeepWalk model from the “DeepWalk: Online Learning of Social Representations”

---

**class** `models.emb.deepwalk.DeepWalk` (*dimension, walk\_length, walk\_num, window\_size, worker, iteration*)

Bases: `models.BaseModel`

The DeepWalk model from the “DeepWalk: Online Learning of Social Representations” paper

**Args:** `hidden_size` (int) : The dimension of node representation. `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `window_size` (int) : The actual context size which is considered in language model. `worker` (int) : The number of workers for word2vec. `iteration` (int) : The number of training iteration in word2vec.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)

**\_walk** (*self, start\_node, walk\_length*)

**\_simulate\_walks** (*self, walk\_length, num\_walks*)

`models.emb.dgk`

## Module Contents

### Classes

---

<i>DeepGraphKernel</i>	The Hin2vec model from the “Deep Graph Kernels”
------------------------	---

---

**class** `models.emb.dgk.DeepGraphKernel` (*hidden\_dim, min\_count, window\_size, sampling\_rate, rounds, epoch, alpha, n\_workers=4*)

Bases: `models.BaseModel`

The Hin2vec model from the “Deep Graph Kernels” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `min_count` (int) : Parameter in word2vec. `window` (int) : The actual context size which is considered in language model. `sampling_rate` (float) : Parameter in word2vec. `iteration` (int) : The number of iteration in WL method. `epoch` (int) : The number of training iteration. `alpha` (float) : The learning rate of word2vec.

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**static feature\_extractor** (*data, rounds, name*)

**static wl\_iterations** (*graph, features, rounds*)

**forward** (*self, graphs, \*\*kwargs*)

**save\_embedding** (*self, output\_path*)

`models.emb.dngr`

## Module Contents

### Classes

---

*DNGR\_layer*


---

<i>DNGR</i>	The DNGR model from the “Deep Neural Networks for Learning Graph Representations”
-------------	---

---

**class** `models.emb.dngr.DNGR_layer` (*num\_node, hidden\_size1, hidden\_size2*)

Bases: `torch.nn.Module`

**forward** (*self, x*)

**class** `models.emb.dngr.DNGR` (*hidden\_size1, hidden\_size2, noise, alpha, step, max\_epoch, lr, cpu*)

Bases: `models.BaseModel`

The DNGR model from the “Deep Neural Networks for Learning Graph Representations” paper

**Args:** `hidden_size1` (int) : The size of the first hidden layer. `hidden_size2` (int) : The size of the second hidden

layer. noise (float) : Denoise rate of DAE. alpha (float) : Parameter in DNGR. step (int) : The max step in random surfing. max\_epoch (int) : The max epoches in training step. lr (float) : Learning rate in DNGR.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**scale\_matrix** (*self, mat*)

**random\_surfing** (*self, adj\_matrix*)

**get\_ppmi\_matrix** (*self, mat*)

**get\_denoised\_matrix** (*self, mat*)

**get\_emb** (*self, matrix*)

**train** (*self, G*)

`models.emb.gatne`

## Module Contents

### Classes

---

*GATNE*

The GATNE model from the [“Representation Learning for Attributed Multiplex Heterogeneous Network”](#)

---

*GATNEModel*

---

*NSLoss*

---

*RWGraph*

---

### Functions

---

*get\_G\_from\_edges*(edges)

---

*generate\_pairs*(all\_walks, vocab, window\_size=5)

---

*generate\_vocab*(all\_walks)

---

*get\_batches*(pairs, neighbors, batch\_size)

---

*generate\_walks*(network\_data, num\_walks, walk\_length, schema=None)

---

**class** `models.emb.gatne.GATNE` (*dimension, walk\_length, walk\_num, window\_size, worker, epoch, batch\_size, edge\_dim, att\_dim, negative\_samples, neighbor\_samples, schema*)

Bases: `models.BaseModel`

The GATNE model from the “Representation Learning for Attributed Multiplex Heterogeneous Network” paper

**Args:** `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `window_size` (int) : The actual context size which is considered in language model. `worker` (int) : The number of workers for word2vec. `epoch` (int) : The number of training epochs. `batch_size` (int) : The size of each training batch. `edge_dim` (int) : Number of edge embedding dimensions. `att_dim` (int) : Number of attention dimensions. `negative_samples` (int) : Negative samples for optimization. `neighbor_samples` (int) : Neighbor samples for aggregation schema (`str`) : The metapath schema used in model. Metapaths are splited with “;”, while each node type are connected with “-” in each metapath. For example:”0-1-0,0-1-2-1-0”

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, network\_data*)

**class** `models.emb.gatne.GATNEModel` (*num\_nodes, embedding\_size, embedding\_u\_size, edge\_type\_count, dim\_a*)

Bases: `torch.nn.Module`

**reset\_parameters** (*self*)

**forward** (*self, train\_inputs, train\_types, node\_neigh*)

**class** `models.emb.gatne.NSLoss` (*num\_nodes, num\_sampled, embedding\_size*)

Bases: `torch.nn.Module`

**reset\_parameters** (*self*)

**forward** (*self, input, embs, label*)

**class** `models.emb.gatne.RWGraph` (*nx\_G, node\_type=None*)

**walk** (*self, walk\_length, start, schema=None*)

**simulate\_walks** (*self, num\_walks, walk\_length, schema=None*)

`models.emb.gatne.get_G_from_edges` (*edges*)

`models.emb.gatne.generate_pairs` (*all\_walks, vocab, window\_size=5*)

`models.emb.gatne.generate_vocab` (*all\_walks*)

`models.emb.gatne.get_batches` (*pairs, neighbors, batch\_size*)

`models.emb.gatne.generate_walks` (*network\_data, num\_walks, walk\_length, schema=None*)

`models.emb.graph2vec`

## Module Contents

### Classes

---

*Graph2Vec*

The Graph2Vec model from the “graph2vec: Learning Distributed Representations of Graphs”

---

**class** `models.emb.graph2vec.Graph2Vec` (*dimension, min\_count, window\_size, dm, sampling\_rate, rounds, epoch, lr, worker=4*)

Bases: `models.BaseModel`

The Graph2Vec model from the “[graph2vec: Learning Distributed Representations of Graphs](#)” paper

**Args:** `hidden_size` (int) : The dimension of node representation. `min_count` (int) : Parameter in doc2vec. `window_size` (int) : The actual context size which is considered in language model. `sampling_rate` (float) : Parameter in doc2vec. `dm` (int) : Parameter in doc2vec. `iteration` (int) : The number of iteration in WL method. `epoch` (int) : The max epoches in training step. `lr` (float) : Learning rate in doc2vec.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**static feature\_extractor** (*data, rounds, name*)

**static wl\_iterations** (*graph, features, rounds*)

**forward** (*self, graphs, \*\*kwargs*)

**save\_embedding** (*self, output\_path*)

`models.emb.grarep`

## Module Contents

### Classes

---

`GraRep`

The GraRep model from the “[Grarep: Learning graph representations with global structural information](#)”

---

**class** `models.emb.grarep.GraRep` (*dimension, step*)

Bases: `models.BaseModel`

The GraRep model from the “[Grarep: Learning graph representations with global structural information](#)” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `step` (int) : The maximum order of transition probability.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)

**\_get\_embedding** (*self, matrix, dimension*)



`models.emb.hin2vec`

## Module Contents

### Classes

---

*Hin2vec\_layer*

---

*RWgraph*

---

*Hin2vec*

The Hin2vec model from the [“HIN2Vec: Explore Meta-paths in Heterogeneous Information Networks for Representation Learning”](#)

---

**class** `models.emb.hin2vec.Hin2vec_layer` (*num\_node, num\_relation, hidden\_size, cpu*)

Bases: `torch.nn.Module`

**regularization** (*self, embr*)

**forward** (*self, x, y, r, l*)

**get\_emb** (*self*)

**class** `models.emb.hin2vec.RWgraph` (*nx\_G, node\_type=None*)

**\_walk** (*self, start\_node, walk\_length*)

**\_simulate\_walks** (*self, walk\_length, num\_walks*)

**data\_preparation** (*self, walks, hop, negative*)

**class** `models.emb.hin2vec.Hin2vec` (*hidden\_dim, walk\_length, walk\_num, batch\_size, hop, negative, epoches, lr, cpu=True*)

Bases: `models.BaseModel`

The Hin2vec model from the [“HIN2Vec: Explore Meta-paths in Heterogeneous Information Networks for Representation Learning”](#) paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `batch_size` (int) : The batch size of training in Hin2vec. `hop` (int) : The number of hop to construct training samples in Hin2vec. `negative` (int) : The number of negative samples for each meta2path pair. `epoches` (int) : The number of training iteration. `lr` (float) : The initial learning rate of SGD. `cpu` (bool) : Use CPU or GPU to train hin2vec.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G, node\_type*)

`models_emb hope`

## Module Contents

### Classes

---

*HOPE*

---

The HOPE model from the “Grarep: Asymmetric transitivity preserving graph embedding”

**class** `models_emb hope.HOPE` (*dimension, beta*)Bases: `models.BaseModel`

The HOPE model from the “Grarep: Asymmetric transitivity preserving graph embedding” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `beta` (float) : Parameter in katz decomposition.**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)The author claim that Katz has superior performance in related tasks  $S_{katz} = (M_g)^{-1} * M_l = (I - \beta * A)^{-1} * \beta * A = (I - \beta * A)^{-1} * (I - (I - \beta * A)) = (I - \beta * A)^{-1} - I$ **\_get\_embedding** (*self, matrix, dimension*)`models_emb line`

## Module Contents

### Classes

---

*LINE*

---

The LINE model from the “Line: Large-scale information network embedding”

**class** `models_emb line.LINE` (*dimension, walk\_length, walk\_num, negative, batch\_size, alpha, order*)Bases: `models.BaseModel`

The LINE model from the “Line: Large-scale information network embedding” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `negative` (int) : The number of negative samples for each edge. `batch_size` (int) : The batch size of training in LINE. `alpha` (float) : The initial learning rate of SGD. `order` (int) : 1 represents perserving 1-st order proximity, 2 represents 2-nd, while 3 means both of them (each of them having dimension/2 node representation).**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

```

train (self, G)
_update (self, vec_u, vec_v, vec_error, label)
_train_line (self, order)

```

## models.emb.metapath2vec

### Module Contents

#### Classes

---

<i>Metapath2vec</i>	The Metapath2vec model from the <a href="#">"metapath2vec: Scalable Representation Learning for Heterogeneous Networks"</a> paper
---------------------	---

---

```

class models.emb.metapath2vec.Metapath2vec (dimension, walk_length, walk_num, window_size, worker, iteration, schema)

```

Bases: *models.BaseModel*

The Metapath2vec model from the ["metapath2vec: Scalable Representation Learning for Heterogeneous Networks"](#) paper

**Args:** *hidden\_size* (int) : The dimension of node representation. *walk\_length* (int) : The walk length. *walk\_num* (int) : The number of walks to sample for each node. *window\_size* (int) : The actual context size which is considered in language model. *worker* (int) : The number of workers for word2vec. *iteration* (int) : The number of training iteration in word2vec. *schema* (str) : The metapath schema used in model. Metapaths are splited with ";", while each node type are connected with "-" in each metapath. For example:"0-1-0,0-2-0,1-0-2-0-1".

```

static add_args (parser)
    Add model-specific arguments to the parser.

```

```

classmethod build_model_from_args (cls, args)
    Build a new model instance.

```

```

train (self, G, node_type)
_walk (self, start_node, walk_length, schema=None)
_simulate_walks (self, walk_length, num_walks, schema='No')

```

## models.emb.netmf

### Module Contents

#### Classes

---

<i>NetMF</i>	The NetMF model from the <a href="#">"Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec"</a>
--------------	--

---

```

class models.emb.netmf.NetMF (dimension, window_size, rank, negative, is_large=False)

```

Bases: *models.BaseModel*

The NetMF model from the “Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `window_size` (int) : The actual context size which is considered in language model. `rank` (int) : The rank in approximate normalized laplacian. `negative` (int) : The number of nagative samples in negative sampling. `is-large` (bool) : When window size is large, use approximated deepwalk matrix to decompose.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)

**\_compute\_deepwalk\_matrix** (*self, A, window, b*)

**\_approximate\_normalized\_laplacian** (*self, A, rank, which='LA'*)

**\_deepwalk\_filter** (*self, evals, window*)

**\_approximate\_deepwalk\_matrix** (*self, evals, D\_rt\_invU, window, vol, b*)

`models.emb.netSMF`

## Module Contents

### Classes

---

*NetSMF*

The NetSMF model from the “NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization”

---

**class** `models.emb.netSMF.NetSMF` (*dimension, window\_size, negative, num\_round, worker*)

Bases: `models.BaseModel`

The NetSMF model from the “NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `window_size` (int) : The actual context size which is considered in language model. `negative` (int) : The number of nagative samples in negative sampling. `num_round` (int) : The number of round in NetSMF. `worker` (int) : The number of workers for NetSMF.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)

**\_get\_embedding\_rand** (*self, matrix*)

**\_path\_sampling** (*self, u, v, r*)

**\_random\_walk\_matrix** (*self, pid*)

`models.emb.node2vec`

## Module Contents

### Classes

---

<code>Node2vec</code>	The node2vec model from the <a href="#">“node2vec: Scalable feature learning for networks”</a>
-----------------------	--

---

**class** `models.emb.node2vec.Node2vec` (*dimension, walk\_length, walk\_num, window\_size, worker, iteration, p, q*)

Bases: `models.BaseModel`

The node2vec model from the [“node2vec: Scalable feature learning for networks”](#) paper

**Args:** `hidden_size` (int) : The dimension of node representation. `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `window_size` (int) : The actual context size which is considered in language model. `worker` (int) : The number of workers for word2vec. `iteration` (int) : The number of training iteration in word2vec. `p` (float) : Parameter in node2vec. `q` (float) : Parameter in node2vec.

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**train** (*self, G*)

**\_node2vec\_walk** (*self, walk\_length, start\_node*)

**\_simulate\_walks** (*self, num\_walks, walk\_length*)

**\_get\_alias\_edge** (*self, src, dst*)

**\_preprocess\_transition\_probs** (*self*)

`models.emb.prone`

## Module Contents

### Classes

---

<code>ProNE</code>	The ProNE model from the <a href="#">“ProNE: Fast and Scalable Network Representation Learning”</a>
--------------------	---

---

**class** `models.emb.prone.ProNE` (*dimension, step, mu, theta*)

Bases: `models.BaseModel`

The ProNE model from the [“ProNE: Fast and Scalable Network Representation Learning”](#) paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `step` (int) : The number of items in the chebyshev expansion. `mu` (float) : Parameter in ProNE. `theta` (float) : Parameter in ProNE.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod** `build_model_from_args` (*cls, args*)

Build a new model instance.

**train** (*self, G*)

`_get_embedding_rand` (*self, matrix*)

`_get_embedding_dense` (*self, matrix, dimension*)

`_pre_factorization` (*self, tran, mask*)

`_chebyshev_gaussian` (*self, A, a, order=5, mu=0.5, s=0.2, plus=False, nn=False*)

`models.emb.pte`

## Module Contents

### Classes

---

*PTE*

The PTE model from the “PTE: Predictive Text Embedding through Large-scale Heterogeneous Text Networks”

---

**class** `models.emb.pte.PTE` (*dimension, walk\_length, walk\_num, negative, batch\_size, alpha*)

Bases: `models.BaseModel`

The PTE model from the “PTE: Predictive Text Embedding through Large-scale Heterogeneous Text Networks” paper.

**Args:** `hidden_size` (int) : The dimension of node representation. `walk_length` (int) : The walk length. `walk_num` (int) : The number of walks to sample for each node. `negative` (int) : The number of negative samples for each edge. `batch_size` (int) : The batch size of training in PTE. `alpha` (float) : The initial learning rate of SGD.

**static** `add_args` (*parser*)

Add model-specific arguments to the parser.

**classmethod** `build_model_from_args` (*cls, args*)

Build a new model instance.

**train** (*self, G, node\_type*)

`_update` (*self, vec\_u, vec\_v, vec\_error, label*)

`_train_line` (*self*)

`models.emb.sdne`

## Module Contents

### Classes

---

*SDNE\_layer*

---

*SDNE*The SDNE model from the “Structural Deep Network Embedding”

---

**class** `models.emb.sdne.SDNE_layer` (*num\_node, hidden\_size1, hidden\_size2, dropout, alpha, beta, nu1, nu2*)Bases: `torch.nn.Module`**forward** (*self, adj\_mat, L\_mat*)**get\_emb** (*self, adj*)**class** `models.emb.sdne.SDNE` (*hidden\_size1, hidden\_size2, dropout, alpha, beta, nu1, nu2, max\_epoch, lr, cpu*)Bases: `models.BaseModel`

The SDNE model from the “Structural Deep Network Embedding” paper

**Args:** `hidden_size1` (int) : The size of the first hidden layer. `hidden_size2` (int) : The size of the second hidden layer. `dropout` (float) : Dropout rate. `alpha` (float) : Trade-off parameter between 1-st and 2-nd order objective function in SDNE. `beta` (float) : Parameter of 2-nd order objective function in SDNE. `nu1` (float) : Parameter of l1 normlization in SDNE. `nu2` (float) : Parameter of l2 normlization in SDNE. `max_epoch` (int) : The max epoches in training step. `lr` (float) : Learning rate in SDNE. `cpu` (bool) : Use CPU or GPU to train `hin2vec`.**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)`models.emb.spectral`

## Module Contents

### Classes

---

*Spectral*

---

The Spectral clustering model from the “Leveraging social media networks for classication”

---

**class** `models.emb.spectral.Spectral` (*dimension*)Bases: `models.BaseModel`

The Spectral clustering model from the “Leveraging social media networks for classication” paper

**Args:** `hidden_size` (int) : The dimension of node representation.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**train** (*self, G*)

`models.nn`

### Submodules

`models.nn.asgcn`

### Module Contents

#### Classes

---

*GraphConvolution*

Simple GCN layer, similar to <https://arxiv.org/abs/1609.02907>

---

*ASGCN*

---

**class** `models.nn.asgcn.GraphConvolution` (*in\_features, out\_features, bias=True*)

Bases: `torch.nn.Module`

Simple GCN layer, similar to <https://arxiv.org/abs/1609.02907>

**reset\_parameters** (*self*)

**forward** (*self, input, adj*)

**\_\_repr\_\_** (*self*)

**class** `models.nn.asgcn.ASGCN` (*num\_features, num\_classes, hidden\_size, num\_layers, dropout, sample\_size*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**reset\_parameters** (*self*)

**set\_adj** (*self, edge\_index, num\_nodes*)

**compute\_adjlist** (*self, sp\_adj, max\_degree=32*)

Transfer sparse adjacent matrix to adj-list format

**from\_adjlist** (*self, adj*)

Transfer adj-list format to sparsensor

**\_sample\_one\_layer** (*self, x, adj, v, sample\_size*)

**sampling** (*self, x, v*)



**forward** (*self*, *x*, *adj*)

`models.nn.dgl_gcc`

## Module Contents

### Classes

---

`NodeClassificationDataset`

---

`GCC`

---

### Functions

---

`batcher()`

---

`test_moco`(*train\_loader*, *model*, *opt*) one epoch training for moco

---

`eigen_decomposition`(*n*, *k*, *laplacian*, *hidden\_size*, *retry*)

---

`_add_undirected_graph_positional_embedding`(*g*, *hidden\_size*, *retry*=10)

---

`_rwr_trace_to_dgl_graph`(*g*, *seed*, *trace*, *positional\_embedding\_size*, *entire\_graph*=False)

---

`models.nn.dgl_gcc.batcher` ()

`models.nn.dgl_gcc.test_moco` (*train\_loader*, *model*, *opt*)  
one epoch training for moco

`models.nn.dgl_gcc.eigen_decomposition` (*n*, *k*, *laplacian*, *hidden\_size*, *retry*)

`models.nn.dgl_gcc._add_undirected_graph_positional_embedding` (*g*, *hidden\_size*, *retry*=10)

`models.nn.dgl_gcc._rwr_trace_to_dgl_graph` (*g*, *seed*, *trace*, *positional\_embedding\_size*, *entire\_graph*=False)

**class** `models.nn.dgl_gcc.NodeClassificationDataset` (*data*, *rw\_hops*=64, *subgraph\_size*=64, *restart\_prob*=0.8, *positional\_embedding\_size*=32, *step\_dist*=[1.0, 0.0, 0.0])

Bases: `object`

`_create_dgl_graph` (*self*, *data*)

`__len__` (*self*)

`_convert_idx` (*self*, *idx*)

`__getitem__` (*self*, *idx*)

**class** `models.nn.dgl_gcc.GCC` (*load\_path*)

Bases: `models.BaseModel`

```
static add_args (parser)  
    Add model-specific arguments to the parser.  
classmethod build_model_from_args (cls, args)  
    Build a new model instance.  
train (self, data)
```

`models.nn.fastgcn`

### Module Contents

#### Classes

---

<code>GraphConvolution</code>	Simple GCN layer, similar to <a href="https://arxiv.org/abs/1609.02907">https://arxiv.org/abs/1609.02907</a> .
<code>FastGCN</code>	

---

```
class models.nn.fastgcn.GraphConvolution (in_features, out_features, bias=True)  
    Bases: torch.nn.Module
```

Simple GCN layer, similar to <https://arxiv.org/abs/1609.02907>

```
reset_parameters (self)  
forward (self, input, adj)  
__repr__ (self)
```

```
class models.nn.fastgcn.FastGCN (num_features, num_classes, hidden_size, num_layers, dropout,  
                                sample_size)
```

Bases: `models.BaseModel`

```
static add_args (parser)  
    Add model-specific arguments to the parser.  
classmethod build_model_from_args (cls, args)  
    Build a new model instance.  
set_adj (self, edge_index, num_nodes)  
_sample_one_layer (self, sampled, sample_size)  
_generate_adj (self, sample1, sample2)  
sampling (self, x, v)  
forward (self, x, adj)
```

`models.nn.gat`

## Module Contents

### Classes

<i>GraphAttentionLayer</i>	Simple GAT layer, similar to <a href="https://arxiv.org/abs/1710.10903">https://arxiv.org/abs/1710.10903</a>
<i>SpecialSpmFunction</i>	Special function for only sparse region backpropataion layer.
<i>SpecialSpm</i>	
<i>SpGraphAttentionLayer</i>	Sparse version GAT layer, similar to <a href="https://arxiv.org/abs/1710.10903">https://arxiv.org/abs/1710.10903</a>
<i>PetarVGAT</i>	
<i>PetarVSpGAT</i>	

**class** `models.nn.gat.GraphAttentionLayer` (*in\_features*, *out\_features*, *dropout*, *alpha*, *concat=True*)

Bases: `torch.nn.Module`

Simple GAT layer, similar to <https://arxiv.org/abs/1710.10903>

**forward** (*self*, *input*, *adj*)

**\_\_repr\_\_** (*self*)

**class** `models.nn.gat.SpecialSpmFunction`

Bases: `torch.autograd.Function`

Special function for only sparse region backpropataion layer.

**static forward** (*ctx*, *indices*, *values*, *shape*, *b*)

**static backward** (*ctx*, *grad\_output*)

**class** `models.nn.gat.SpecialSpm`

Bases: `torch.nn.Module`

**forward** (*self*, *indices*, *values*, *shape*, *b*)

**class** `models.nn.gat.SpGraphAttentionLayer` (*in\_features*, *out\_features*, *dropout*, *alpha*, *concat=True*)

Bases: `torch.nn.Module`

Sparse version GAT layer, similar to <https://arxiv.org/abs/1710.10903>

**forward** (*self*, *input*, *edge*)

**\_\_repr\_\_** (*self*)

**class** `models.nn.gat.PetarVGAT` (*nfeat*, *nhid*, *nclass*, *dropout*, *alpha*, *nheads*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod** `build_model_from_args` (*cls, args*)  
 Build a new model instance.

**forward** (*self, x, adj*)

**class** `models.nn.gat.PetarVSpGAT` (*nfeat, nhid, nclass, dropout, alpha, nheads*)  
 Bases: `models.nn.gat.PetarVGAT`

**forward** (*self, x, adj*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.gcn`

## Module Contents

### Classes

---

<code>GraphConvolution</code>	Simple GCN layer, similar to <a href="https://arxiv.org/abs/1609.02907">https://arxiv.org/abs/1609.02907</a>
<code>TKipfGCN</code>	

---

**class** `models.nn.gcn.GraphConvolution` (*in\_features, out\_features, bias=True*)  
 Bases: `torch.nn.Module`

Simple GCN layer, similar to <https://arxiv.org/abs/1609.02907>

**reset\_parameters** (*self*)

**forward** (*self, input, edge\_index*)

**\_\_repr\_\_** (*self*)

**class** `models.nn.gcn.TKipfGCN` (*nfeat, nhid, nclass, dropout*)  
 Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod** `build_model_from_args` (*cls, args*)  
 Build a new model instance.

**forward** (*self, x, adj*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.graphsage`

## Module Contents

### Classes

---

*Graphsage*

---

**class** `models.nn.graphsage.Graphsage` (*num\_features, num\_classes, hidden\_size, num\_layers, sample\_size, dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**sampler** (*self, edge\_index, num\_sample*)

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.mixhop`

## Module Contents

### Classes

---

*MixHop*

---

**class** `models.nn.mixhop.MixHop` (*num\_features, num\_classes, hidden\_size, num\_layers, dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.mlp`

## Module Contents

### Classes

---

*MLP*

---

**class** `models.nn.mlp.MLP` (*num\_features, num\_classes, hidden\_size, num\_layers, dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.patchy_san`

## Module Contents

### Classes

---

*PatchySAN*

The Patchy-SAN model from the “Learning Convolutional Neural Networks for Graphs”

---

### Functions

---

*assemble\_neighbor*(*G, node, num\_neighbor, assemble\_neighbors* for node with BFS strategy  
*sorted\_nodes*)

*cmp*(*s1, s2*)

---

*one\_dim\_wl*(*graph\_list, init\_labels, iteration=5*) 1-dimension WI method used for node normalization for  
all the subgraphs

---

*node\_selection\_with\_1d\_wl*(*G, features, num\_channel, num\_sample, num\_neighbor, stride*) construct features for cnn

---

*get\_single\_feature*(*data, num\_features, num\_classes, num\_sample, num\_neighbor, stride=1*) construct features

---

**class** `models.nn.patchy_san.PatchySAN` (*batch\_size, num\_features, num\_classes, num\_sample, stride, num\_neighbor, iteration*)

Bases: `models.BaseModel`

The Patchy-SAN model from the “Learning Convolutional Neural Networks for Graphs” paper.

**Args:** `batch_size` (int) : The batch size of training. `sample` (int) : Number of chosen vertexes. `stride` (int) : Node selection stride. `neighbor` (int) : The number of neighbor for each node. `iteration` (int) : The number of training iteration.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**classmethod split\_dataset** (*self, dataset, args*)

**build\_model** (*self, num\_channel, num\_sample, num\_neighbor, num\_class*)

**forward** (*self, batch*)

`models.nn.patchy_san.assemble_neighbor` (*G, node, num\_neighbor, sorted\_nodes*)

assemble neighbors for node with BFS strategy

`models.nn.patchy_san.cmp` (*s1, s2*)

`models.nn.patchy_san.one_dim_wl` (*graph\_list, init\_labels, iteration=5*)

1-dimension WL method used for node normalization for all the subgraphs

`models.nn.patchy_san.node_selection_with_1d_wl` (*G, features, num\_channel, num\_sample, num\_neighbor, stride*)

construct features for cnn

`models.nn.patchy_san.get_single_feature` (*data, num\_features, num\_classes, num\_sample, num\_neighbor, stride=1*)

construct features

`models.nn.pyg_cheb`

## Module Contents

### Classes

---

*Chebyshev*

---

**class** `models.nn.pyg_cheb.Chebyshev` (*num\_features, num\_classes, hidden\_size, num\_layers, dropout, filter\_size*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.pyg_dgcnn`

## Module Contents

### Classes

---

<i>DGCNN</i>	EdgeConv and DynamicGraph in paper <a href="#">“Dynamic Graph CNN for Learning on</a>
--------------	---

---

**class** `models.nn.pyg_dgcnn.DGCNN` (*in\_feats*, *hidden\_dim*, *out\_feats*, *k=20*, *dropout=0.5*)

Bases: `models.BaseModel`

EdgeConv and DynamicGraph in paper [“Dynamic Graph CNN for Learning on Point Clouds”](https://arxiv.org/pdf/1801.07829.pdf) <<https://arxiv.org/pdf/1801.07829.pdf>>\_\_.

**in\_feats** [int] Size of each input sample.

**out\_feats** [int] Size of each output sample.

**hidden\_dim** [int] Dimension of hidden layer embedding.

**k** [int] Number of nearest neighbors.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)

Build a new model instance.

**classmethod split\_dataset** (*cls*, *dataset*, *args*)

**forward** (*self*, *batch*)

`models.nn.pyg_diffpool`

## Module Contents

### Classes

---

<i>EntropyLoss</i>	
<i>LinkPredLoss</i>	
<i>GraphSAGE</i>	GraphSAGE from <a href="#">“Inductive Representation Learning on Large Graphs”</a> .
<i>BatchedGraphSAGE</i>	GraphSAGE with mini-batch
<i>BatchedDiffPoolLayer</i>	DIFFPOOL from paper <a href="#">“Hierarchical Graph Representation Learning</a>
<i>BatchedDiffPool</i>	DIFFPOOL layer with batch forward
<i>DiffPool</i>	DIFFPOOL from paper <a href="#">“Hierarchical Graph Representation Learning</a>

---



---

## Functions

---

```
toBatchedGraph(batch_adj,          batch_feat,
node_per_pool_graph)
```

---

```
class models.nn.pyg_diffpool.EntropyLoss
```

```
    Bases: torch.nn.Module
```

```
    forward (self, adj, anext, s_l)
```

```
class models.nn.pyg_diffpool.LinkPredLoss
```

```
    Bases: torch.nn.Module
```

```
    forward (self, adj, anext, s_l)
```

```
class models.nn.pyg_diffpool.GraphSAGE (in_feats, hidden_dim, out_feats, num_layers,
                                         dropout=0.5, normalize=False, concat=False,
                                         use_bn=False)
```

```
    Bases: torch.nn.Module
```

GraphSAGE from “Inductive Representation Learning on Large Graphs”.

```
..math:: h^{i+1}_{\mathcal{N}(v)} = \text{AGGREGATE}_{\{k\}}(h_{\mathcal{U}}^k) \oplus h_{\mathcal{N}(v)}^k =
\text{sigma}(\mathbf{W}^k \cdot \text{CONCAT}(h_{\mathcal{V}}^k, h_{\mathcal{N}(v)}^k))
```

**Args:** *in\_feats* (int) : Size of each input sample. *hidden\_dim* (int) : Size of hidden layer dimension. *out\_feats* (int) : Size of each output sample. *num\_layers* (int) : Number of GraphSAGE Layers. *dropout* (float, optional) : Size of dropout, default: 0.5. *normalize* (bool, optional) : Normalize features after each layer if True, default: True.

```
    forward (self, x, edge_index, edge_weight=None)
```

```
class models.nn.pyg_diffpool.BatchedGraphSAGE (in_feats, out_feats, use_bn=True,
                                                self_loop=True)
```

```
    Bases: torch.nn.Module
```

GraphSAGE with mini-batch

**Args:** *in\_feats* (int) : Size of each input sample. *out\_feats* (int) : Size of each output sample. *use\_bn* (bool) : Apply batch normalization if True, default: True. *self\_loop* (bool) : Add self loop if True, default: True.

```
    forward (self, x, adj)
```

```
class models.nn.pyg_diffpool.BatchedDiffPoolLayer (in_feats, out_feats, assign_dim,
                                                  batch_size, dropout=0.5,
                                                  link_pred_loss=True, entropy_loss=True)
```

```
    Bases: torch.nn.Module
```

DIFFPOOL from paper “Hierarchical Graph Representation Learning with Differentiable Pooling”.

$$X^{(l+1)} = S^{(l)T} Z^{(l)} A^{(l+1)} = S^{(l)T} A^{(l)} S^{(l)} Z^{(l)} = \text{GNN}_{l, \text{embed}}(A^{(l)}, X^{(l)}) S^{(l)} = \text{softmax}(\text{GNN}_{l, \text{pool}}(A^{(l)}, X^{(l)}))$$

**in\_feats** [int] Size of each input sample.

**out\_feats** [int] Size of each output sample.

**assign\_dim** [int] Size of next adjacency matrix.

**batch\_size** [int] Size of each mini-batch.

**dropout** [float, optional] Size of dropout, default: 0.5.

**link\_pred\_loss** [bool, optional] Use link prediction loss if True, default: True.

**forward** (*self*, *x*, *edge\_index*, *batch*, *edge\_weight=None*)

**get\_loss** (*self*)

**class** `models.nn.pyg_diffpool.BatchedDiffPool` (*in\_feats*, *next\_size*, *emb\_size*, *use\_bn=True*,  
*self\_loop=True*, *use\_link\_loss=False*,  
*use\_entropy=True*)

Bases: `torch.nn.Module`

DIFFPOOL layer with batch forward

**in\_feats** [int] Size of each input sample.

**next\_size** [int] Size of next adjacency matrix.

**emb\_size** [int] Dimension of next node feature matrix.

**use\_bn** [bool, optional] Apply batch normalization if True, default: True.

**self\_loop** [bool, optional] Add self loop if True, default: True.

**use\_link\_loss** [bool, optional] Use link prediction loss if True, default: True.

**use\_entropy** [bool, optional] Use entropy prediction loss if True, default: True.

**forward** (*self*, *x*, *adj*)

**get\_loss** (*self*)

`models.nn.pyg_diffpool.toBatchedGraph` (*batch\_adj*, *batch\_feat*, *node\_per\_pool\_graph*)

**class** `models.nn.pyg_diffpool.DiffPool` (*in\_feats*, *hidden\_dim*, *embed\_dim*, *num\_classes*,  
*num\_layers*, *num\_pool\_layers*, *assign\_dim*,  
*pooling\_ratio*, *batch\_size*, *dropout=0.5*,  
*no\_link\_pred=True*, *concat=False*, *use\_bn=False*)

Bases: `models.BaseModel`

DIFFPOOL from paper [Hierarchical Graph Representation Learning with Differentiable Pooling](#).

**in\_feats** [int] Size of each input sample.

**hidden\_dim** [int] Size of hidden layer dimension of GNN.

**embed\_dim** [int] Size of embedded node feature, output size of GNN.

**num\_classes** [int] Number of target classes.

**num\_layers** [int] Number of GNN layers.

**num\_pool\_layers** [int] Number of pooling.

**assign\_dim** [int] Embedding size after the first pooling.

**pooling\_ratio** [float] Size of each pooling ratio.

**batch\_size** [int] Size of each mini-batch.

**dropout** [float, optional] Size of dropout, default: 0.5.

**no\_link\_pred** [bool, optional] If True, use link prediction loss, default: True.

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)

Build a new model instance.

```

classmethod split_dataset (cls, dataset, args)
reset_parameters (self)
after_pooling_forward (self, gnn_layers, adj, x, concat=False)
forward (self, batch)
loss (self, prediction, label)

```

`models.nn.pyg_drgat`

## Module Contents

### Classes

---

*DrGAT*

---

```

class models.nn.pyg_drgat.DrGAT (num_features, num_classes, hidden_size, num_heads,
                                dropout)
    Bases: models.BaseModel
    static add_args (parser)
        Add model-specific arguments to the parser.
    classmethod build_model_from_args (cls, args)
        Build a new model instance.
    forward (self, x, edge_index)
    loss (self, data)
    predict (self, data)

```

`models.nn.pyg_drgcn`

## Module Contents

### Classes

---

*DrGCN*

---

```

class models.nn.pyg_drgcn.DrGCN (num_features, num_classes, hidden_size, num_layers,
                                dropout)
    Bases: models.BaseModel
    static add_args (parser)
        Add model-specific arguments to the parser.
    classmethod build_model_from_args (cls, args)
        Build a new model instance.
    forward (self, x, edge_index)

```

**loss** (*self*, *data*)

**predict** (*self*, *data*)

`models.nn.pyg_gat`

### Module Contents

#### Classes

---

*GAT*

---

**class** `models.nn.pyg_gat.GAT` (*num\_features*, *num\_classes*, *hidden\_size*, *num\_heads*, *dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)

Build a new model instance.

**forward** (*self*, *x*, *edge\_index*)

**loss** (*self*, *data*)

**predict** (*self*, *data*)

`models.nn.pyg_gcn`

### Module Contents

#### Classes

---

*GCN*

---

**class** `models.nn.pyg_gcn.GCN` (*num\_features*, *num\_classes*, *hidden\_size*, *num\_layers*, *dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)

Build a new model instance.

**forward** (*self*, *x*, *edge\_index*)

**loss** (*self*, *data*)

**predict** (*self*, *data*)

`models.nn.pyg_gin`

## Module Contents

### Classes

<code>GINLayer</code>	Graph Isomorphism Network layer from paper “How Powerful are Graph
<code>GINMLP</code>	Multilayer perception with batch normalization
<code>GIN</code>	Graph Isomorphism Network from paper “How Powerful are Graph

**class** `models.nn.pyg_gin.GINLayer` (*apply\_func=None, eps=0, train\_eps=True*)  
 Bases: `torch.nn.Module`

Graph Isomorphism Network layer from paper “How Powerful are Graph Neural Networks?”.

$$h_i^{(l+1)} = f_{\Theta} \left( (1 + \epsilon) h_i^l + \text{sum} \left( \{ h_j^l, j \in \mathcal{N}(i) \} \right) \right)$$

**apply\_func** [callable layer function)] layer or function applied to update node feature

**eps** [float32, optional] Initial *epsilon* value.

**train\_eps** [bool, optional] If True, *epsilon* will be a learnable parameter.

**forward** (*self, x, edge\_index, edge\_weight=None*)

**class** `models.nn.pyg_gin.GINMLP` (*in\_feats, out\_feats, hidden\_dim, num\_layers, use\_bn=True, activation=None*)

Bases: `torch.nn.Module`

Multilayer perception with batch normalization

$$x^{(i+1)} = \sigma(W^i x^{(i)})$$

**in\_feats** [int] Size of each input sample.

**out\_feats** [int] Size of each output sample.

**hidden\_dim** [int] Size of hidden layer dimension.

**use\_bn** [bool, optional] Apply batch normalization if True, default: `True`).

**forward** (*self, x*)

**class** `models.nn.pyg_gin.GIN` (*num\_layers, in\_feats, out\_feats, hidden\_dim, num\_mlp\_layers, eps=0, pooling='sum', train\_eps=False, dropout=0.5*)

Bases: `models.BaseModel`

Graph Isomorphism Network from paper “How Powerful are Graph Neural Networks?”.

**Args:**

**num\_layers** [int] Number of GIN layers

**in\_feats** [int] Size of each input sample

**out\_feats** [int] Size of each output sample

**hidden\_dim** [int] Size of each hidden layer dimension

**num\_mlp\_layers** [int] Number of MLP layers

**eps** [float32, optional] Initial *epsilon* value, default: 0

**pooling** [str, optional] Aggregator type to use, default: sum

**train\_eps** [bool, optional] If True, *epsilon* will be a learnable parameter, default: True

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**classmethod split\_dataset** (*cls, dataset, args*)

**forward** (*self, batch*)

**loss** (*self, output, label=None*)

`models.nn.pyg_gtn`

### Module Contents

#### Classes

---

*GTConv*

---

*GTLayer*

---

*GTN*

---

**class** `models.nn.pyg_gtn.GTConv` (*in\_channels, out\_channels, num\_nodes*)  
Bases: `torch.nn.Module`

**reset\_parameters** (*self*)

**forward** (*self, A*)

**class** `models.nn.pyg_gtn.GTLayer` (*in\_channels, out\_channels, num\_nodes, first=True*)  
Bases: `torch.nn.Module`

**forward** (*self, A, H\_=None*)

**class** `models.nn.pyg_gtn.GTN` (*num\_edge, num\_channels, w\_in, w\_out, num\_class, num\_nodes, num\_layers*)  
Bases: `models.BaseModel`

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
Build a new model instance.

**normalization** (*self, H*)

**norm** (*self, edge\_index, num\_nodes, edge\_weight, improved=False, dtype=None*)

**forward** (*self, A, X, target\_x, target*)

**loss** (*self, data*)

**evaluate** (*self, data, nodes, targets*)

`models.nn.pyg_han`

## Module Contents

### Classes

---

*AttentionLayer*

---

*HANLayer*

---

*HAN*

---

**class** `models.nn.pyg_han.AttentionLayer` (*num\_features*)

Bases: `torch.nn.Module`

**forward** (*self, x*)

**class** `models.nn.pyg_han.HANLayer` (*num\_edge, w\_in, w\_out*)

Bases: `torch.nn.Module`

**forward** (*self, x, adj*)

**class** `models.nn.pyg_han.HAN` (*num\_edge, w\_in, w\_out, num\_class, num\_nodes, num\_layers*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**forward** (*self, A, X, target\_x, target*)

**loss** (*self, data*)

**evaluate** (*self, data, nodes, targets*)

`models.nn.pyg_infograph`

## Module Contents

### Classes

---

*SUPEncoder*

Encoder used in supervised model with Set2set in paper  
 “Order Matters: Sequence to sequence for sets”

---

*Encoder*

Encoder stacked with GIN layers

---

*FF*

Residual MLP layers.

---

continues on next page

Table 88 – continued from previous page

<i>InfoGraph</i>	Implimentation of Infograph in paper <a href="#">”InfoGraph: Un-supervised and Semi-supervised Graph-Level Representation</a>
------------------	---

---

**class** `models.nn.pyg_infograph.SUPEncoder` (*num\_features, dim, num\_layers=1*)  
 Bases: `torch.nn.Module`

Encoder used in supervised model with Set2set in paper “*Order Matters: Sequence to sequence for sets*” <<https://arxiv.org/abs/1511.06391>> and NNConv in paper “*Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs*” <<https://arxiv.org/abs/1704.02901>>

**forward** (*self, x, edge\_index, batch, edge\_attr*)

**class** `models.nn.pyg_infograph.Encoder` (*in\_feats, hidden\_dim, num\_layers=3, num\_mlp\_layers=2, pooling='sum'*)  
 Bases: `torch.nn.Module`

Encoder stacked with GIN layers

**in\_feats** [int] Size of each input sample.

**hidden\_feats** [int] Size of output embedding.

**num\_layers** [int, optional] Number of GIN layers, default: 3.

**num\_mlp\_layers** [int, optional] Number of MLP layers for each GIN layer, default: 2.

**pooling** [str, optional] Aggragation type, default : sum.

**forward** (*self, x, edge\_index, batch, \*args*)

**class** `models.nn.pyg_infograph.FF` (*in\_feats, out\_feats*)  
 Bases: `torch.nn.Module`

Residual MLP layers.

**..math::**  $out = \text{mathbf{MLP}}(x) + \text{mathbf{Linear}}(x)$

**in\_feats** [int] Size of each input sample

**out\_feats** [int] Size of each output sample

**forward** (*self, x*)

**class** `models.nn.pyg_infograph.InfoGraph` (*in\_feats, hidden\_dim, out\_feats, num\_layers=3, unsup=True*)  
 Bases: `models.BaseModel`

**Implimentation of Infograph in paper [”InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation Learning via Mutual Information Maximization](#)”** <<https://openreview.net/forum?id=r1lfF2NYvH>>\_`

**in\_feats** [int] Size of each input sample.

**out\_feats** [int] Size of each output sample.

**num\_layers** [int, optional] Number of MLP layers in encoder, default: 3.

**unsup** [bool, optional] Use unsupervised model if True, default: True.

**static add\_args** (*parser*)  
 Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)  
 Build a new model instance.



```

classmethod split_dataset (cls, dataset, args)
reset_parameters (self)
forward (self, batch)
sup_forward (self, x, edge_index=None, batch=None, label=None, edge_attr=None)
unsup_forward (self, x, edge_index=None, batch=None)
sup_loss (self, prediction, label=None)
unsup_loss (self, x, edge_index=None, batch=None)
unsup_sup_loss (self, x, edge_index, batch)
static mi_loss (pos_mask, neg_mask, mi, pos_div, neg_div)

```

`models.nn.pyg_infomax`

## Module Contents

### Classes

---

*Encoder*

---

*Infomax*

---

### Functions

---

*corruption*(*x, edge\_index*)

---

**class** `models.nn.pyg_infomax.Encoder` (*in\_channels, hidden\_channels*)

Bases: `torch.nn.Module`

**forward** (*self, x, edge\_index*)

`models.nn.pyg_infomax.corruption` (*x, edge\_index*)

**class** `models.nn.pyg_infomax.Infomax` (*num\_features, num\_classes, hidden\_size*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

`models.nn.pyg_sortpool`

## Module Contents

### Classes

---

<code>SortPool</code>	Implimentation of sortpooling in paper “An End-to-End Deep Learning
-----------------------	---

---

### Functions

---

<code>scatter_sum(src, index, dim, dim_size)</code>
<code>sparse2dense_batch(x, batch=None, fill_value=0)</code>

---

`models.nn.pyg_sortpool.scatter_sum(src, index, dim, dim_size)`

`models.nn.pyg_sortpool.sparse2dense_batch(x, batch=None, fill_value=0)`

**class** `models.nn.pyg_sortpool.SortPool` (*in\_feats*, *hidden\_dim*, *num\_classes*, *num\_layers*, *out\_channel*, *kernel\_size*, *k=30*, *dropout=0.5*)

Bases: `models.BaseModel`

Implimentation of sortpooling in paper “An End-to-End Deep Learning Architecture for Graph Classification” <[https://www.cse.wustl.edu/~muhan/papers/AAAI\\_2018\\_DGCNN.pdf](https://www.cse.wustl.edu/~muhan/papers/AAAI_2018_DGCNN.pdf)>\_\_.

**in\_feats** [int] Size of each input sample.

**out\_feats** [int] Size of each output sample.

**hidden\_dim** [int] Dimension of hidden layer embedding.

**num\_classes** [int] Number of target classes.

**num\_layers** [int] Number of graph neural network layers before pooling.

**k** [int, optional] Number of selected features to sort, default: 30.

**out\_channel** [int] Number of the first convolution’s output channels.

**kernel\_size** [int] Size of the first convolution’s kernel.

**dropout** [float, optional] Size of dropout, default: 0.5.

**static add\_args** (*parser*)  
Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)  
Build a new model instance.

**classmethod split\_dataset** (*cls*, *dataset*, *args*)

**forward** (*self*, *batch*)

`models.nn.pyg_srgcn`

## Module Contents

### Classes

---

*NodeAdaptiveEncoder*

---

*SrgcnHead*

---

*SrgcnSoftmaxHead*

---

*SRGCN*

---

**class** `models.nn.pyg_srgcn.NodeAdaptiveEncoder` (*num\_features*, *dropout*=0.5)

Bases: `nn.Module`

**forward** (*self*, *x*)

**class** `models.nn.pyg_srgcn.SrgcnHead` (*num\_features*, *out\_feats*, *attention*, *activation*, *normalization*, *nhop*, *subheads*=2, *dropout*=0.5, *node\_dropout*=0.5, *alpha*=0.2, *concat*=True)

Bases: `nn.Module`

**forward** (*self*, *x*, *edge\_index*, *edge\_attr*)

**class** `models.nn.pyg_srgcn.SrgcnSoftmaxHead` (*num\_features*, *out\_feats*, *attention*, *activation*, *nhop*, *normalization*, *dropout*=0.5, *node\_dropout*=0.5, *alpha*=0.2)

Bases: `nn.Module`

**forward** (*self*, *x*, *edge\_index*, *edge\_attr*)

**class** `models.nn.pyg_srgcn.SRGCN` (*num\_features*, *hidden\_size*, *num\_classes*, *attention*, *activation*, *nhop*, *normalization*, *dropout*, *node\_dropout*, *alpha*, *nhead*, *subheads*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls*, *args*)

Build a new model instance.

**forward** (*self*, *batch*)

**loss** (*self*, *data*)

**predict** (*self*, *data*)

`models.nn.pyg_unet`

### Module Contents

#### Classes

---

*UNet*

---

**class** `models.nn.pyg_unet.UNet` (*num\_features, num\_classes, hidden\_size, num\_layers, dropout*)

Bases: `models.BaseModel`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

**forward** (*self, x, edge\_index*)

**loss** (*self, data*)

**predict** (*self, data*)

### 6.7.2 Submodules

`models.base_model`

### Module Contents

#### Classes

---

*BaseModel*

---

**class** `models.base_model.BaseModel`

Bases: `torch.nn.Module`

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**abstract classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

## 6.7.3 Package Contents

### Classes

---

*BaseModel*

---

### Functions

---

<i>register_model</i> (name)	New model types can be added to cogdl with the <i>register_model()</i>
<i>alias_setup</i> (probs)	Compute utility lists for non-uniform sampling from discrete distributions.
<i>alias_draw</i> (J, q)	Draw sample from a non-uniform discrete distribution using alias sampling.
<i>build_model</i> (args)	

---

#### **class** models.BaseModel

Bases: torch.nn.Module

**static add\_args** (*parser*)

Add model-specific arguments to the parser.

**abstract classmethod build\_model\_from\_args** (*cls, args*)

Build a new model instance.

models.pyg = False

models.dgl\_import = False

models.MODEL\_REGISTRY

models.register\_model (*name*)

New model types can be added to cogdl with the *register\_model()* function decorator.

For example:

```
@register_model('gat')
class GAT(BaseModel):
    (...)
```

**Args:** name (str): the name of the model

models.alias\_setup (*probs*)

Compute utility lists for non-uniform sampling from discrete distributions. Refer to <https://hips.seas.harvard.edu/blog/2013/03/03/the-alias-method-efficient-sampling-with-many-discrete-outcomes/> for details

models.alias\_draw (*J, q*)

Draw sample from a non-uniform discrete distribution using alias sampling.

models.model\_name

models.build\_model (*args*)



## INDICES AND TABLES

- genindex
- modindex
- search





## PYTHON MODULE INDEX

### d

- data, 30
- data.batch, 30
- data.data, 31
- data.dataloader, 33
- data.dataset, 34
- data.download, 35
- data.extract, 35
- data.makedirs, 36
- datasets, 47
- datasets.gatne, 47
- datasets.gcc\_data, 48
- datasets.gtn\_data, 49
- datasets.han\_data, 50
- datasets.matlab\_matrix, 51
- datasets.pyg, 53
- datasets.pyg\_modelnet, 54

### l

- layers, 24
- layers.gcc\_module, 24
- layers.maggregator, 26
- layers.mixhop\_layer, 27
- layers.se\_layer, 27
- layers.srgcn\_module, 28

### m

- models, 56
- models.base\_model, 88
- models.emb, 56
- models.emb.deepwalk, 56
- models.emb.dgk, 57
- models.emb.dngr, 57
- models.emb.gatne, 58
- models.emb.graph2vec, 59
- models.emb.grarep, 60
- models.emb.hin2vec, 61
- models.emb.hope, 62
- models.emb.line, 62
- models.emb.metapath2vec, 63
- models.emb.netmf, 63
- models.emb.netsmf, 64

- models.emb.node2vec, 65
- models.emb.prone, 65
- models.emb.pte, 66
- models.emb.sdne, 67
- models.emb.spectral, 67
- models.nn, 68
- models.nn.asgcn, 68
- models.nn.dgl\_gcc, 69
- models.nn.fastgcn, 70
- models.nn.gat, 71
- models.nn.gcn, 72
- models.nn.graphsage, 73
- models.nn.mixhop, 73
- models.nn.mlp, 74
- models.nn.patchy\_san, 74
- models.nn.pyg\_cheb, 75
- models.nn.pyg\_dgcnn, 76
- models.nn.pyg\_diffpool, 76
- models.nn.pyg\_drgat, 79
- models.nn.pyg\_drgcn, 79
- models.nn.pyg\_gat, 80
- models.nn.pyg\_gcn, 80
- models.nn.pyg\_gin, 81
- models.nn.pyg\_gtn, 82
- models.nn.pyg\_han, 83
- models.nn.pyg\_infograph, 83
- models.nn.pyg\_infomax, 85
- models.nn.pyg\_sortpool, 86
- models.nn.pyg\_srgcn, 87
- models.nn.pyg\_unet, 88

### o

- options, 23

### t

- tasks, 41
- tasks.base\_task, 41
- tasks.graph\_classification, 41
- tasks.heterogeneous\_node\_classification, 42
- tasks.link\_prediction, 42
- tasks.multiplex\_link\_prediction, 43

tasks.multiplex\_node\_classification, 44  
tasks.node\_classification, 44  
tasks.node\_classification\_sampling, 44  
tasks.unsupervised\_graph\_classification,  
45  
tasks.unsupervised\_node\_classification,  
45

## U

utils, 24

## Symbols

- `__call__()` (*data.Data method*), 37
- `__call__()` (*data.data.Data method*), 31
- `__contains__()` (*data.Data method*), 37
- `__contains__()` (*data.data.Data method*), 31
- `__getitem__()` (*data.Data method*), 37
- `__getitem__()` (*data.Dataset method*), 39
- `__getitem__()` (*data.data.Data method*), 31
- `__getitem__()` (*data.dataset.Dataset method*), 35
- `__getitem__()` (*datasets.pyg.ENZYMES method*), 54
- `__getitem__()` (*datasets.pyg\_modelnet.ModelNetData10 method*), 55
- `__getitem__()` (*datasets.pyg\_modelnet.ModelNetData40 method*), 55
- `__getitem__()` (*models.nn.dgl\_gcc.NodeClassificationDataset method*), 69
- `__inc__()` (*data.Data method*), 37
- `__inc__()` (*data.data.Data method*), 32
- `__iter__()` (*data.Data method*), 37
- `__iter__()` (*data.data.Data method*), 31
- `__len__()` (*data.Data method*), 37
- `__len__()` (*data.Dataset method*), 39
- `__len__()` (*data.data.Data method*), 31
- `__len__()` (*data.dataset.Dataset method*), 34
- `__len__()` (*datasets.pyg\_modelnet.ModelNetData10 method*), 55
- `__len__()` (*datasets.pyg\_modelnet.ModelNetData40 method*), 55
- `__len__()` (*models.nn.dgl\_gcc.NodeClassificationDataset method*), 69
- `__repr__()` (*data.Data method*), 38
- `__repr__()` (*data.Dataset method*), 39
- `__repr__()` (*data.data.Data method*), 32
- `__repr__()` (*data.dataset.Dataset method*), 35
- `__repr__()` (*datasets.gatme.GatmeDataset method*), 47
- `__repr__()` (*datasets.gtn\_data.GTNDataset method*), 49
- `__repr__()` (*datasets.han\_data.HANDataset method*), 51
- `__repr__()` (*layers.MeanAggregator method*), 30
- `__repr__()` (*layers.maggregator.MeanAggregator method*), 27
- `__repr__()` (*models.nn.asgcn.GraphConvolution method*), 68
- `__repr__()` (*models.nn.fastgcn.GraphConvolution method*), 70
- `__repr__()` (*models.nn.gat.GraphAttentionLayer method*), 71
- `__repr__()` (*models.nn.gat.SpGraphAttentionLayer method*), 71
- `__repr__()` (*models.nn.gcn.GraphConvolution method*), 72
- `__setitem__()` (*data.Data method*), 37
- `__setitem__()` (*data.data.Data method*), 31
- `__add_undirected_graph_positional_embedding()` (*in module models.nn.dgl\_gcc*), 69
- `__approximate_deepwalk_matrix()` (*models.emb.netmf.NetMF method*), 64
- `__approximate_normalized_laplacian()` (*models.emb.netmf.NetMF method*), 64
- `__chebyshev_gaussian()` (*models.emb.prone.ProNE method*), 66
- `__compute_deepwalk_matrix()` (*models.emb.netmf.NetMF method*), 64
- `__convert_idx()` (*models.nn.dgl\_gcc.NodeClassificationDataset method*), 69
- `__create_dgl_graph()` (*models.nn.dgl\_gcc.NodeClassificationDataset method*), 69
- `__deepwalk_filter()` (*models.emb.netmf.NetMF method*), 64
- `__download()` (*data.Dataset method*), 39
- `__download()` (*data.dataset.Dataset method*), 35
- `__evaluate()` (*tasks.unsupervised\_graph\_classification.UnsupervisedGraphClassification method*), 45
- `__evaluate()` (*tasks.unsupervised\_node\_classification.UnsupervisedNodeClassification method*), 46
- `__generate_adj()` (*models.nn.fastgcn.FastGCN method*), 70
- `__get_alias_edge()` (*models.emb.node2vec.Node2vec method*), 65



`add_args()` (*models.emb.spectral.Spectral static method*), 68  
`add_args()` (*models.nn.asgcn.ASGCN static method*), 68  
`add_args()` (*models.nn.dgl\_gcc.GCC static method*), 69  
`add_args()` (*models.nn.fastgcn.FastGCN static method*), 70  
`add_args()` (*models.nn.gat.PetarVGAT static method*), 71  
`add_args()` (*models.nn.gcn.TKipfGCN static method*), 72  
`add_args()` (*models.nn.graphsage.Graphsage static method*), 73  
`add_args()` (*models.nn.mixhop.MixHop static method*), 73  
`add_args()` (*models.nn.mlp.MLP static method*), 74  
`add_args()` (*models.nn.patchy\_san.PatchySAN static method*), 75  
`add_args()` (*models.nn.pyg\_cheb.Chebyshev static method*), 75  
`add_args()` (*models.nn.pyg\_dgcnn.DGCNN static method*), 76  
`add_args()` (*models.nn.pyg\_diffpool.DiffPool static method*), 78  
`add_args()` (*models.nn.pyg\_drgat.DrGAT static method*), 79  
`add_args()` (*models.nn.pyg\_drgcn.DrGCN static method*), 79  
`add_args()` (*models.nn.pyg\_gat.GAT static method*), 80  
`add_args()` (*models.nn.pyg\_gcn.GCN static method*), 80  
`add_args()` (*models.nn.pyg\_gin.GIN static method*), 82  
`add_args()` (*models.nn.pyg\_gtn.GTN static method*), 82  
`add_args()` (*models.nn.pyg\_han.HAN static method*), 83  
`add_args()` (*models.nn.pyg\_infograph.InfoGraph static method*), 84  
`add_args()` (*models.nn.pyg\_infomax.Infomax static method*), 85  
`add_args()` (*models.nn.pyg\_sortpool.SortPool static method*), 86  
`add_args()` (*models.nn.pyg\_srcn.SRGCN static method*), 87  
`add_args()` (*models.nn.pyg\_unet.UNet static method*), 88  
`add_args()` (*tasks.base\_task.BaseTask static method*), 41  
`add_args()` (*tasks.BaseTask static method*), 46  
`add_args()` (*tasks.graph\_classification.GraphClassification static method*), 41  
`add_args()` (*tasks.heterogeneous\_node\_classification.HeterogeneousNode static method*), 42  
`add_args()` (*tasks.link\_prediction.LinkPrediction static method*), 43  
`add_args()` (*tasks.multiplex\_link\_prediction.MultiplexLinkPrediction static method*), 43  
`add_args()` (*tasks.multiplex\_node\_classification.MultiplexNodeClassification static method*), 44  
`add_args()` (*tasks.node\_classification.NodeClassification static method*), 44  
`add_args()` (*tasks.node\_classification\_sampling.NodeClassificationSampling static method*), 45  
`add_args()` (*tasks.unsupervised\_graph\_classification.UnsupervisedGraphClassification static method*), 45  
`add_args()` (*tasks.unsupervised\_node\_classification.UnsupervisedNodeClassification static method*), 46  
`add_dataset_args()` (*in module options*), 23  
`add_model_args()` (*in module options*), 23  
`add_remaining_self_loops()` (*in module utils*), 24  
`add_task_args()` (*in module options*), 23  
`adj_pow_x()` (*layers.mixhop\_layer.MixHopLayer method*), 27  
`adj_pow_x()` (*layers.MixHopLayer method*), 30  
`after_pooling_forward()` (*models.nn.pyg\_diffpool.DiffPool method*), 79  
`alias_draw()` (*in module models*), 89  
`alias_setup()` (*in module models*), 89  
`AmazonDataset` (*class in datasets.gatne*), 47  
`apply()` (*data.Data method*), 38  
`apply()` (*data.data.Data method*), 32  
`apply_to_device()` (*datasets.gtn\_data.GTNDataset method*), 49  
`apply_to_device()` (*datasets.han\_data.HANDataset method*), 51  
`ApplyNodeFunc` (*class in layers.gcc\_module*), 25  
`ArgClass` (*class in utils*), 24  
`args` (*in module utils*), 24  
`ASGCN` (*class in models.nn.asgcn*), 68  
`assemble_neighbor()` (*in module models.nn.patchy\_san*), 75  
`AttentionLayer` (*class in models.nn.pyg\_han*), 83

## B

`backward()` (*models.nn.gat.SpecialSpmFunction static method*), 71  
`BaseModel` (*class in models*), 89  
`BaseModel` (*class in models.base\_model*), 88  
`BaseTask` (*class in tasks*), 46  
`BaseTask` (*class in tasks.base\_task*), 41  
`Batch` (*class in data*), 38  
`Batch` (*class in data.batch*), 30

BatchedDiffPool (class in *models.nn.pyg\_diffpool*), 78  
 BatchedDiffPoolLayer (class in *models.nn.pyg\_diffpool*), 77  
 BatchedGraphSAGE (class in *models.nn.pyg\_diffpool*), 77  
 batcher() (in module *models.nn.dgl\_gcc*), 69  
 BlogcatalogDataset (class in *datasets.matlab\_matrix*), 52  
 build\_args\_from\_dict() (in module *utils*), 24  
 build\_dataset() (in module *datasets*), 56  
 build\_dataset\_from\_name() (in module *datasets*), 56  
 build\_model() (in module *models*), 89  
 build\_model() (*models.nn.patchy\_san.PatchySAN* class method), 75  
 build\_model\_from\_args() (*models.base\_model.BaseModel* class method), 88  
 build\_model\_from\_args() (*models.BaseModel* class method), 89  
 build\_model\_from\_args() (*models.emb.deepwalk.DeepWalk* class method), 56  
 build\_model\_from\_args() (*models.emb.dgk.DeepGraphKernel* class method), 57  
 build\_model\_from\_args() (*models.emb.dngr.DNGR* class method), 58  
 build\_model\_from\_args() (*models.emb.gatne.GATNE* class method), 59  
 build\_model\_from\_args() (*models.emb.graph2vec.Graph2Vec* class method), 60  
 build\_model\_from\_args() (*models.emb.grarep.GraRep* class method), 60  
 build\_model\_from\_args() (*models.emb.hin2vec.Hin2vec* class method), 61  
 build\_model\_from\_args() (*models.emb.hope.HOPE* class method), 62  
 build\_model\_from\_args() (*models.emb.line.LINE* class method), 62  
 build\_model\_from\_args() (*models.emb.metapath2vec.Metapath2vec* class method), 63  
 build\_model\_from\_args() (*models.emb.netmf.NetMF* class method), 64  
 build\_model\_from\_args() (*models.emb.netsmf.NetSMF* class method), 64  
 build\_model\_from\_args() (*models.emb.node2vec.Node2vec* class method), 65  
 build\_model\_from\_args() (*models.emb.prone.ProNE* class method), 66  
 build\_model\_from\_args() (*models.emb.pte.PTE* class method), 66  
 build\_model\_from\_args() (*models.emb.sdne.SDNE* class method), 67  
 build\_model\_from\_args() (*models.emb.spectral.Spectral* class method), 68  
 build\_model\_from\_args() (*models.nn.asgcn.ASGCN* class method), 68  
 build\_model\_from\_args() (*models.nn.dgl\_gcc.GCC* class method), 70  
 build\_model\_from\_args() (*models.nn.fastgcn.FastGCN* class method), 70  
 build\_model\_from\_args() (*models.nn.gat.PetarVGAT* class method), 71  
 build\_model\_from\_args() (*models.nn.gcn.TKipfGCN* class method), 72  
 build\_model\_from\_args() (*models.nn.graphsage.Graphsage* class method), 73  
 build\_model\_from\_args() (*models.nn.mixhop.MixHop* class method), 73  
 build\_model\_from\_args() (*models.nn.mlp.MLP* class method), 74  
 build\_model\_from\_args() (*models.nn.patchy\_san.PatchySAN* class method), 75  
 build\_model\_from\_args() (*models.nn.pyg\_cheb.Chebyshev* class method), 75  
 build\_model\_from\_args() (*models.nn.pyg\_dgcnn.DGCNN* class method), 76  
 build\_model\_from\_args() (*models.nn.pyg\_diffpool.DiffPool* class method), 78  
 build\_model\_from\_args() (*models.nn.pyg\_drgat.DrGAT* class method), 79  
 build\_model\_from\_args() (*models.nn.pyg\_drgcn.DrGCN* class method), 79  
 build\_model\_from\_args() (*models.nn.pyg\_gat.GAT* class method), 80  
 build\_model\_from\_args() (*models.nn.pyg\_gcn.GCN* class method), 80  
 build\_model\_from\_args() (*models.nn.pyg\_gin.GIN* class method), 82  
 build\_model\_from\_args() (*models.nn.pyg\_gtn.GTN* class method), 82  
 build\_model\_from\_args() (*models.nn.pyg\_han.HAN* class method), 83  
 build\_model\_from\_args() (mod-



- els.nn.pyg\_infograph.InfoGraph* class method), 84
- `build_model_from_args()` (*models.nn.pyg\_infomax.Infomax* class method), 85
- `build_model_from_args()` (*models.nn.pyg\_sortpool.SortPool* class method), 86
- `build_model_from_args()` (*models.nn.pyg\_srgcn.SRGCN* class method), 87
- `build_model_from_args()` (*models.nn.pyg\_unet.UNet* class method), 88
- `build_task()` (in module *tasks*), 47
- ## C
- `cat_dim()` (*data.Data* method), 37
- `cat_dim()` (*data.data.Data* method), 32
- Chebyshev* (class in *models.nn.pyg\_cheb*), 75
- CiteSeerDataset* (class in *datasets.pyg*), 53
- `clone()` (*data.Data* method), 38
- `clone()` (*data.data.Data* method), 32
- `cmp()` (in module *models.nn.patchy\_san*), 75
- CollabDataset* (class in *datasets.pyg*), 54
- ColumnUniform* (class in *layers.srgcn\_module*), 29
- `compute_adjlist()` (*models.nn.asgcn.ASGCN* method), 68
- `contiguous()` (*data.Data* method), 38
- `contiguous()` (*data.data.Data* method), 32
- CoraDataset* (class in *datasets.pyg*), 53
- `corruption()` (in module *models.nn.pyg\_infomax*), 85
- `cuda()` (*data.Data* method), 38
- `cuda()` (*data.data.Data* method), 32
- `cumsum()` (*data.Batch* method), 38
- `cumsum()` (*data.batch.Batch* method), 30
- ## D
- data*  
module, 30
- Data* (class in *data*), 36
- Data* (class in *data.data*), 31
- data.batch*  
module, 30
- data.data*  
module, 31
- data.data\_loader*  
module, 33
- data.dataset*  
module, 34
- data.download*  
module, 35
- data.extract*  
module, 35
- data.makedirs*  
module, 36
- `data_preparation()` (*models.emb.hin2vec.RWgraph* method), 61
- DataListLoader* (class in *data*), 40
- DataListLoader* (class in *data.data\_loader*), 33
- DataLoader* (class in *data*), 39
- DataLoader* (class in *data.data\_loader*), 33
- Dataset* (class in *data*), 38
- Dataset* (class in *data.dataset*), 34
- `dataset_name` (in module *datasets*), 55
- DATASET\_REGISTRY* (in module *datasets*), 55
- datasets*  
module, 47
- datasets.gatne*  
module, 47
- datasets.gcc\_data*  
module, 48
- datasets.gtn\_data*  
module, 49
- datasets.han\_data*  
module, 50
- datasets.matlab\_matrix*  
module, 51
- datasets.pyg*  
module, 53
- datasets.pyg\_modelnet*  
module, 54
- DBLP\_GTNDataset* (class in *datasets.gtn\_data*), 50
- DBLP\_HANDataset* (class in *datasets.han\_data*), 51
- DeepGraphKernel* (class in *models.emb.dgk*), 57
- DeepWalk* (class in *models.emb.deepwalk*), 56
- DenseDataLoader* (class in *data*), 40
- DenseDataLoader* (class in *data.data\_loader*), 33
- DGCNN* (class in *models.nn.pyg\_dgcnn*), 76
- `dgl_import` (in module *datasets*), 55
- `dgl_import` (in module *models*), 89
- DiffPool* (class in *models.nn.pyg\_diffpool*), 78
- `divide_data()` (in module *tasks.link\_prediction*), 43
- DNGR* (class in *models.emb.dngr*), 57
- DNGR\_layer* (class in *models.emb.dngr*), 57
- `download()` (*data.Dataset* method), 39
- `download()` (*data.dataset.Dataset* method), 34
- `download()` (*datasets.gatne.GatneDataset* method), 47
- `download()` (*datasets.gcc\_data.EdgeList* method), 48
- `download()` (*datasets.gtn\_data.GTNDataset* method), 49
- `download()` (*datasets.han\_data.HANDataset* method), 51
- `download()` (*datasets.matlab\_matrix.MatlabMatrix* method), 52
- `download_url()` (in module *data*), 40
- `download_url()` (in module *data.download*), 35

DrGAT (*class in models.nn.pyg\_drgat*), 79

DrGCN (*class in models.nn.pyg\_drgcn*), 79

## E

EdgeAttention (*class in layers.srgcn\_module*), 28

Edgelist (*class in datasets.gcc\_data*), 48

eigen\_decomposition() (*in module models.nn.dgl\_gcc*), 69

Encoder (*class in models.nn.pyg\_infograph*), 84

Encoder (*class in models.nn.pyg\_infomax*), 85

enhance\_emb() (*tasks.unsupervised\_node\_classification.UnsupervisedNodeClassification method*), 46

EntropyLoss (*class in models.nn.pyg\_diffpool*), 77

ENZYMES (*class in datasets.pyg*), 54

evaluate() (*in module tasks.link\_prediction*), 43

evaluate() (*in module tasks.multiplex\_link\_prediction*), 43

evaluate() (*models.nn.pyg\_gtn.GTN method*), 82

evaluate() (*models.nn.pyg\_han.HAN method*), 83

extract\_bz2() (*in module data*), 40

extract\_bz2() (*in module data.extract*), 36

extract\_gz() (*in module data*), 40

extract\_gz() (*in module data.extract*), 36

extract\_tar() (*in module data*), 40

extract\_tar() (*in module data.extract*), 35

extract\_zip() (*in module data*), 40

extract\_zip() (*in module data.extract*), 36

## F

FastGCN (*class in models.nn.fastgcn*), 70

feature\_extractor() (*models.emb.dgk.DeepGraphKernel static method*), 57

feature\_extractor() (*models.emb.graph2vec.Graph2Vec static method*), 60

FF (*class in models.nn.pyg\_infograph*), 84

files\_exist() (*in module data.dataset*), 34

FlickrDataset (*class in datasets.matlab\_matrix*), 52

forward() (*layers.gcc\_module.ApplyNodeFunc method*), 25

forward() (*layers.gcc\_module.GraphEncoder method*), 26

forward() (*layers.gcc\_module.MLP method*), 25

forward() (*layers.gcc\_module.SELayer method*), 25

forward() (*layers.gcc\_module.UnsupervisedGAT method*), 25

forward() (*layers.gcc\_module.UnsupervisedGIN method*), 26

forward() (*layers.gcc\_module.UnsupervisedMPNN method*), 25

forward() (*layers.maggregator.MeanAggregator method*), 27

forward() (*layers.MeanAggregator method*), 29

forward() (*layers.mixhop\_layer.MixHopLayer method*), 27

forward() (*layers.MixHopLayer method*), 30

forward() (*layers.se\_layer.SELayer method*), 27

forward() (*layers.SELayer method*), 30

forward() (*layers.srgcn\_module.ColumnUniform method*), 29

forward() (*layers.srgcn\_module.EdgeAttention method*), 28

forward() (*layers.srgcn\_module.Gaussian method*), 29

forward() (*layers.srgcn\_module.HeatKernel method*), 29

forward() (*layers.srgcn\_module.Identity method*), 28

forward() (*layers.srgcn\_module.NodeAttention method*), 28

forward() (*layers.srgcn\_module.NormIdentity method*), 29

forward() (*layers.srgcn\_module.PPR method*), 29

forward() (*layers.srgcn\_module.RowSoftmax method*), 29

forward() (*layers.srgcn\_module.RowUniform method*), 29

forward() (*layers.srgcn\_module.SymmetryNorm method*), 29

forward() (*models.emb.dgk.DeepGraphKernel method*), 57

forward() (*models.emb.dngr.DNGR\_layer method*), 57

forward() (*models.emb.gatne.GATNEModel method*), 59

forward() (*models.emb.gatne.NSLoss method*), 59

forward() (*models.emb.graph2vec.Graph2Vec method*), 60

forward() (*models.emb.hin2vec.Hin2vec\_layer method*), 61

forward() (*models.emb.sdne.SDNE\_layer method*), 67

forward() (*models.nn.asgcn.ASGCN method*), 68

forward() (*models.nn.asgcn.GraphConvolution method*), 68

forward() (*models.nn.fastgcn.FastGCN method*), 70

forward() (*models.nn.fastgcn.GraphConvolution method*), 70

forward() (*models.nn.gat.GraphAttentionLayer method*), 71

forward() (*models.nn.gat.PetarVGAT method*), 72

forward() (*models.nn.gat.PetarVSpGAT method*), 72

forward() (*models.nn.gat.SpecialSpm method*), 71

forward() (*models.nn.gat.SpecialSpmFunction static method*), 71

forward() (*models.nn.gat.SpGraphAttentionLayer method*), 71

forward() (*models.nn.gcn.GraphConvolution method*), 72



- forward() (*models.nn.gcn.TKipfGCN method*), 72
- forward() (*models.nn.graphsage.Graphsage method*), 73
- forward() (*models.nn.mixhop.MixHop method*), 73
- forward() (*models.nn.mlp.MLP method*), 74
- forward() (*models.nn.patchy\_san.PatchySAN method*), 75
- forward() (*models.nn.pyg\_cheb.Chebyshev method*), 75
- forward() (*models.nn.pyg\_dgcnn.DGCNN method*), 76
- forward() (*models.nn.pyg\_diffpool.BatchedDiffPool method*), 78
- forward() (*models.nn.pyg\_diffpool.BatchedDiffPoolLayer method*), 78
- forward() (*models.nn.pyg\_diffpool.BatchedGraphSAGE method*), 77
- forward() (*models.nn.pyg\_diffpool.DiffPool method*), 79
- forward() (*models.nn.pyg\_diffpool.EntropyLoss method*), 77
- forward() (*models.nn.pyg\_diffpool.GraphSAGE method*), 77
- forward() (*models.nn.pyg\_diffpool.LinkPredLoss method*), 77
- forward() (*models.nn.pyg\_drgat.DrGAT method*), 79
- forward() (*models.nn.pyg\_drgcn.DrGCN method*), 79
- forward() (*models.nn.pyg\_gat.GAT method*), 80
- forward() (*models.nn.pyg\_gcn.GCN method*), 80
- forward() (*models.nn.pyg\_gin.GIN method*), 82
- forward() (*models.nn.pyg\_gin.GINLayer method*), 81
- forward() (*models.nn.pyg\_gin.GINMLP method*), 81
- forward() (*models.nn.pyg\_gtn.GTConv method*), 82
- forward() (*models.nn.pyg\_gtn.GTLayer method*), 82
- forward() (*models.nn.pyg\_gtn.GTN method*), 82
- forward() (*models.nn.pyg\_han.AttentionLayer method*), 83
- forward() (*models.nn.pyg\_han.HAN method*), 83
- forward() (*models.nn.pyg\_han.HANLayer method*), 83
- forward() (*models.nn.pyg\_infograph.Encoder method*), 84
- forward() (*models.nn.pyg\_infograph.FF method*), 84
- forward() (*models.nn.pyg\_infograph.InfoGraph method*), 85
- forward() (*models.nn.pyg\_infograph.SUPEncoder method*), 84
- forward() (*models.nn.pyg\_infomax.Encoder method*), 85
- forward() (*models.nn.pyg\_infomax.Infomax method*), 85
- forward() (*models.nn.pyg\_sortpool.SortPool method*), 86
- forward() (*models.nn.pyg\_srgcn.NodeAdaptiveEncoder method*), 87
- forward() (*models.nn.pyg\_srgcn.SRGCN method*), 87
- forward() (*models.nn.pyg\_srgcn.SrgcnHead method*), 87
- forward() (*models.nn.pyg\_srgcn.SrgcnSoftmaxHead method*), 87
- forward() (*models.nn.pyg\_unet.UNet method*), 88
- from\_adjlist() (*models.nn.asgcn.ASGCN method*), 68
- from\_data\_list() (*data.Batch static method*), 38
- from\_data\_list() (*data.batch.Batch static method*), 30
- from\_dict() (*data.Data static method*), 37
- from\_dict() (*data.data.Data static method*), 31
- ## G
- GAT (*class in models.nn.pyg\_gat*), 80
- GATNE (*class in models.emb.gatne*), 58
- GatneDataset (*class in datasets.gatne*), 47
- GATNEModel (*class in models.emb.gatne*), 59
- Gaussian (*class in layers.srgcn\_module*), 28
- GCC (*class in models.nn.dgl\_gcc*), 69
- GCN (*class in models.nn.pyg\_gcn*), 80
- gen\_node\_pairs() (*in module tasks.link\_prediction*), 43
- generate\_data() (*tasks.graph\_classification.GraphClassification method*), 42
- generate\_pairs() (*in module models.emb.gatne*), 59
- generate\_vocab() (*in module models.emb.gatne*), 59
- generate\_walks() (*in module models.emb.gatne*), 59
- get() (*data.Dataset method*), 39
- get() (*data.dataset.Dataset method*), 34
- get() (*datasets.gatne.GatneDataset method*), 47
- get() (*datasets.gcc\_data.Edgelist method*), 48
- get() (*datasets.gtn\_data.GTNDataset method*), 49
- get() (*datasets.han\_data.HANDataset method*), 51
- get() (*datasets.matlab\_matrix.MatlabMatrix method*), 52
- get\_all() (*datasets.pyg\_modelnet.ModelNetData10 method*), 55
- get\_all() (*datasets.pyg\_modelnet.ModelNetData40 method*), 55
- get\_batches() (*in module models.emb.gatne*), 59
- get\_batches() (*in module tasks.node\_classification\_sampling*), 45
- get\_denoised\_matrix() (*models.emb.dngr.DNGR method*), 58
- get\_display\_data\_parser() (*in module options*), 23
- get\_download\_data\_parser() (*in module options*), 23

get\_emb() (*models.emb.dngr.DNGR method*), 58  
 get\_emb() (*models.emb.hin2vec.Hin2vec\_layer method*), 61  
 get\_emb() (*models.emb.sdne.SDNE\_layer method*), 67  
 get\_G\_from\_edges() (*in module models.emb.gatne*), 59  
 get\_loss() (*models.nn.pyg\_diffpool.BatchedDiffPool method*), 78  
 get\_loss() (*models.nn.pyg\_diffpool.BatchedDiffPoolLayer method*), 78  
 get\_parser() (*in module options*), 23  
 get\_ppmi\_matrix() (*models.emb.dngr.DNGR method*), 58  
 get\_score() (*in module tasks.link\_prediction*), 43  
 get\_score() (*in module tasks.multiplex\_link\_prediction*), 43  
 get\_single\_feature() (*in module models.nn.patchy\_san*), 75  
 get\_training\_parser() (*in module options*), 23  
 GIN (*class in models.nn.pyg\_gin*), 81  
 GINLayer (*class in models.nn.pyg\_gin*), 81  
 GINMLP (*class in models.nn.pyg\_gin*), 81  
 Graph2Vec (*class in models.emb.graph2vec*), 59  
 GraphAttentionLayer (*class in models.nn.gat*), 71  
 GraphClassification (*class in tasks.graph\_classification*), 41  
 GraphConvolution (*class in models.nn.asgcn*), 68  
 GraphConvolution (*class in models.nn.fastgcn*), 70  
 GraphConvolution (*class in models.nn.gcn*), 72  
 GraphEncoder (*class in layers.gcc\_module*), 26  
 Graphsage (*class in models.nn.graphsage*), 73  
 GraphSAGE (*class in models.nn.pyg\_diffpool*), 77  
 GraRep (*class in models.emb.grarep*), 60  
 GTConv (*class in models.nn.pyg\_gtn*), 82  
 GTLayer (*class in models.nn.pyg\_gtn*), 82  
 GTN (*class in models.nn.pyg\_gtn*), 82  
 GTNDataset (*class in datasets.gtn\_data*), 49

## H

HAN (*class in models.nn.pyg\_han*), 83  
 HANDataset (*class in datasets.han\_data*), 50  
 HANLayer (*class in models.nn.pyg\_han*), 83  
 HeatKernel (*class in layers.srgcn\_module*), 29  
 HeterogeneousNodeClassification (*class in tasks.heterogeneous\_node\_classification*), 42  
 Hin2vec (*class in models.emb.hin2vec*), 61  
 Hin2vec\_layer (*class in models.emb.hin2vec*), 61  
 HOPE (*class in models.emb.hope*), 62

## I

Identity (*class in layers.srgcn\_module*), 28  
 IMDB\_GTNDataset (*class in datasets.gtn\_data*), 50  
 IMDB\_HANDataset (*class in datasets.han\_data*), 51  
 ImdbBinaryDataset (*class in datasets.pyg*), 54

ImdbMultiDataset (*class in datasets.pyg*), 54  
 InfoGraph (*class in models.nn.pyg\_infograph*), 84  
 Infomax (*class in models.nn.pyg\_infomax*), 85  
 is\_coalesced() (*data.Data method*), 38  
 is\_coalesced() (*data.data.Data method*), 32

## K

keys() (*data.Data property*), 37  
 keys() (*data.data.Data property*), 31

## L

layer (*in module layers.mixhop\_layer*), 27  
 layers  
   module, 24  
 layers.gcc\_module  
   module, 24  
 layers.maggregator  
   module, 26  
 layers.mixhop\_layer  
   module, 27  
 layers.se\_layer  
   module, 27  
 layers.srgcn\_module  
   module, 28  
 LINE (*class in models.emb.line*), 62  
 LinkPrediction (*class in tasks.link\_prediction*), 43  
 LinkPredLoss (*class in models.nn.pyg\_diffpool*), 77  
 loss() (*models.nn.gat.PetarVSpGAT method*), 72  
 loss() (*models.nn.gcn.TKipfGCN method*), 72  
 loss() (*models.nn.graphsage.Graphsage method*), 73  
 loss() (*models.nn.mixhop.MixHop method*), 73  
 loss() (*models.nn.mlp.MLP method*), 74  
 loss() (*models.nn.pyg\_cheb.Chebyshev method*), 75  
 loss() (*models.nn.pyg\_diffpool.DiffPool method*), 79  
 loss() (*models.nn.pyg\_drgat.DrGAT method*), 79  
 loss() (*models.nn.pyg\_drgcn.DrGCN method*), 79  
 loss() (*models.nn.pyg\_gat.GAT method*), 80  
 loss() (*models.nn.pyg\_gcn.GCN method*), 80  
 loss() (*models.nn.pyg\_gin.GIN method*), 82  
 loss() (*models.nn.pyg\_gtn.GTN method*), 82  
 loss() (*models.nn.pyg\_han.HAN method*), 83  
 loss() (*models.nn.pyg\_infomax.Infomax method*), 85  
 loss() (*models.nn.pyg\_srgcn.SRGCN method*), 87  
 loss() (*models.nn.pyg\_unet.UNet method*), 88

## M

makedirs() (*in module data.makedirs*), 36  
 MatlabMatrix (*class in datasets.matlab\_matrix*), 52  
 maybe\_log() (*in module data.extract*), 35  
 MeanAggregator (*class in layers*), 29  
 MeanAggregator (*class in layers.maggregator*), 26  
 Metapath2vec (*class in models.emb.metapath2vec*), 63

`mi_loss()` (*models.nn.pyg\_infograph.InfoGraph static method*), 85  
`MixHop` (*class in models.nn.mixhop*), 73  
`MixHopLayer` (*class in layers*), 30  
`MixHopLayer` (*class in layers.mixhop\_layer*), 27  
`MLP` (*class in layers.gcc\_module*), 25  
`MLP` (*class in models.nn.mlp*), 74  
`model_name` (*in module models*), 89  
`MODEL_REGISTRY` (*in module models*), 89  
`ModelNet10` (*class in datasets.pyg\_modelnet*), 54  
`ModelNet40` (*class in datasets.pyg\_modelnet*), 55  
`ModelNetData10` (*class in datasets.pyg\_modelnet*), 55  
`ModelNetData40` (*class in datasets.pyg\_modelnet*), 55  
`models`  
  module, 56  
`models.base_model`  
  module, 88  
`models.emb`  
  module, 56  
`models.emb.deepwalk`  
  module, 56  
`models.emb.dgk`  
  module, 57  
`models.emb.dngr`  
  module, 57  
`models.emb.gatne`  
  module, 58  
`models.emb.graph2vec`  
  module, 59  
`models.emb.grarep`  
  module, 60  
`models.emb.hin2vec`  
  module, 61  
`models.emb.hope`  
  module, 62  
`models.emb.line`  
  module, 62  
`models.emb.metapath2vec`  
  module, 63  
`models.emb.netmf`  
  module, 63  
`models.emb.netsmf`  
  module, 64  
`models.emb.node2vec`  
  module, 65  
`models.emb.prone`  
  module, 65  
`models.emb.pte`  
  module, 66  
`models.emb.sdne`  
  module, 67  
`models.emb.spectral`  
  module, 67  
`models.nn`  
  module, 68  
`models.nn.asgcn`  
  module, 68  
`models.nn.dgl_gcc`  
  module, 69  
`models.nn.fastgcn`  
  module, 70  
`models.nn.gat`  
  module, 71  
`models.nn.gcn`  
  module, 72  
`models.nn.graphsage`  
  module, 73  
`models.nn.mixhop`  
  module, 73  
`models.nn.mlp`  
  module, 74  
`models.nn.patchy_san`  
  module, 74  
`models.nn.pyg_cheb`  
  module, 75  
`models.nn.pyg_dgcnn`  
  module, 76  
`models.nn.pyg_diffpool`  
  module, 76  
`models.nn.pyg_drgat`  
  module, 79  
`models.nn.pyg_drgcn`  
  module, 79  
`models.nn.pyg_gat`  
  module, 80  
`models.nn.pyg_gcn`  
  module, 80  
`models.nn.pyg_gin`  
  module, 81  
`models.nn.pyg_gtn`  
  module, 82  
`models.nn.pyg_han`  
  module, 83  
`models.nn.pyg_infograph`  
  module, 83  
`models.nn.pyg_infomax`  
  module, 85  
`models.nn.pyg_sortpool`  
  module, 86  
`models.nn.pyg_srgcn`  
  module, 87  
`models.nn.pyg_unet`  
  module, 88  
  module  
    data, 30  
    data.batch, 30

data.data, 31  
 data.dataloader, 33  
 data.dataset, 34  
 data.download, 35  
 data.extract, 35  
 data.makedirs, 36  
 datasets, 47  
 datasets.gatne, 47  
 datasets.gcc\_data, 48  
 datasets.gtn\_data, 49  
 datasets.han\_data, 50  
 datasets.matlab\_matrix, 51  
 datasets.pyg, 53  
 datasets.pyg\_modelnet, 54  
 layers, 24  
 layers.gcc\_module, 24  
 layers.maggregator, 26  
 layers.mixhop\_layer, 27  
 layers.se\_layer, 27  
 layers.srgcn\_module, 28  
 models, 56  
 models.base\_model, 88  
 models.emb, 56  
 models.emb.deepwalk, 56  
 models.emb.dgk, 57  
 models.emb.dngr, 57  
 models.emb.gatne, 58  
 models.emb.graph2vec, 59  
 models.emb.grarep, 60  
 models.emb.hin2vec, 61  
 models.emb.hope, 62  
 models.emb.line, 62  
 models.emb.metapath2vec, 63  
 models.emb.netmf, 63  
 models.emb.netsmf, 64  
 models.emb.node2vec, 65  
 models.emb.prone, 65  
 models.emb.pte, 66  
 models.emb.sdne, 67  
 models.emb.spectral, 67  
 models.nn, 68  
 models.nn.asgcn, 68  
 models.nn.dgl\_gcc, 69  
 models.nn.fastgcn, 70  
 models.nn.gat, 71  
 models.nn.gcn, 72  
 models.nn.graphsage, 73  
 models.nn.mixhop, 73  
 models.nn.mlp, 74  
 models.nn.patchy\_san, 74  
 models.nn.pyg\_cheb, 75  
 models.nn.pyg\_dgcnn, 76  
 models.nn.pyg\_diffpool, 76  
 models.nn.pyg\_drgat, 79  
 models.nn.pyg\_drgcn, 79  
 models.nn.pyg\_gat, 80  
 models.nn.pyg\_gcn, 80  
 models.nn.pyg\_gin, 81  
 models.nn.pyg\_gtn, 82  
 models.nn.pyg\_han, 83  
 models.nn.pyg\_infograph, 83  
 models.nn.pyg\_infomax, 85  
 models.nn.pyg\_sortpool, 86  
 models.nn.pyg\_srgcn, 87  
 models.nn.pyg\_unet, 88  
 options, 23  
 tasks, 41  
 tasks.base\_task, 41  
 tasks.graph\_classification, 41  
 tasks.heterogeneous\_node\_classification, 42  
 tasks.link\_prediction, 42  
 tasks.multiplex\_link\_prediction, 43  
 tasks.multiplex\_node\_classification, 44  
 tasks.node\_classification, 44  
 tasks.node\_classification\_sampling, 44  
 tasks.unsupervised\_graph\_classification, 45  
 tasks.unsupervised\_node\_classification, 45  
 utils, 24  
 MultiplexLinkPrediction (class in *tasks.multiplex\_link\_prediction*), 43  
 MultiplexNodeClassification (class in *tasks.multiplex\_node\_classification*), 44  
 MUTAGDataset (class in *datasets.pyg*), 54

## N

NCT109Dataset (class in *datasets.pyg*), 54  
 NCT1Dataset (class in *datasets.pyg*), 54  
 NetMF (class in *models.emb.netmf*), 63  
 NetSMF (class in *models.emb.netsmf*), 64  
 Node2vec (class in *models.emb.node2vec*), 65  
 node\_degree\_as\_feature() (in module *tasks.graph\_classification*), 41  
 node\_selection\_with\_1d\_wl() (in module *models.nn.patchy\_san*), 75  
 NodeAdaptiveEncoder (class in *models.nn.pyg\_srgcn*), 87  
 NodeAttention (class in *layers.srgcn\_module*), 28  
 NodeClassification (class in *tasks.node\_classification*), 44  
 NodeClassificationDataset (class in *models.nn.dgl\_gcc*), 69  
 NodeClassificationSampling (class in *tasks.node\_classification\_sampling*), 45

- norm() (*layers.maggregator.MeanAggregator static method*), 27
- norm() (*layers.MeanAggregator static method*), 29
- norm() (*models.nn.pyg\_gtn.GTN method*), 82
- normalization() (*models.nn.pyg\_gtn.GTN method*), 82
- NormIdentity (*class in layers.srgcn\_module*), 29
- NSLoss (*class in models.emb.gatne*), 59
- num\_edges() (*data.Data property*), 38
- num\_edges() (*data.data.Data property*), 32
- num\_features() (*data.Data property*), 38
- num\_features() (*data.data.Data property*), 32
- num\_features() (*data.Dataset property*), 39
- num\_features() (*data.dataset.Dataset property*), 34
- num\_graphs() (*data.Batch property*), 38
- num\_graphs() (*data.batch.Batch property*), 31
- num\_nodes() (*data.Data property*), 38
- num\_nodes() (*data.data.Data property*), 32
- ## O
- one\_dim\_wl() (*in module models.nn.patchy\_san*), 75
- options  
module, 23
- ## P
- parse\_args\_and\_arch() (*in module options*), 23
- PatchySAN (*class in models.nn.patchy\_san*), 74
- PetarVGAT (*class in models.nn.gat*), 71
- PetarVSpGAT (*class in models.nn.gat*), 72
- PPIDataset (*class in datasets.matlab\_matrix*), 52
- PPR (*class in layers.srgcn\_module*), 29
- predict() (*models.nn.gat.PetarVSpGAT method*), 72
- predict() (*models.nn.gcn.TKipfGCN method*), 72
- predict() (*models.nn.graphsage.Graphsage method*), 73
- predict() (*models.nn.mixhop.MixHop method*), 73
- predict() (*models.nn.mlp.MLP method*), 74
- predict() (*models.nn.pyg\_cheb.Chebyshev method*), 75
- predict() (*models.nn.pyg\_drgat.DrGAT method*), 79
- predict() (*models.nn.pyg\_drgcn.DrGCN method*), 80
- predict() (*models.nn.pyg\_gat.GAT method*), 80
- predict() (*models.nn.pyg\_gcn.GCN method*), 80
- predict() (*models.nn.pyg\_infomax.Infomax method*), 85
- predict() (*models.nn.pyg\_srgcn.SRGCN method*), 87
- predict() (*models.nn.pyg\_unet.UNet method*), 88
- predict() (*tasks.unsupervised\_node\_classification.TopKRanker method*), 46
- process() (*data.Dataset method*), 39
- process() (*data.dataset.Dataset method*), 34
- process() (*datasets.gatne.GatneDataset method*), 47
- process() (*datasets.gcc\_data.Edgelist method*), 48
- process() (*datasets.gtn\_data.GTNDataset method*), 49
- process() (*datasets.han\_data.HANDataset method*), 51
- process() (*datasets.matlab\_matrix.MatlabMatrix method*), 52
- processed\_file\_names() (*data.Dataset property*), 39
- processed\_file\_names() (*data.dataset.Dataset property*), 34
- processed\_file\_names() (*datasets.gatne.GatneDataset property*), 47
- processed\_file\_names() (*datasets.gcc\_data.Edgelist property*), 48
- processed\_file\_names() (*datasets.gtn\_data.GTNDataset property*), 49
- processed\_file\_names() (*datasets.han\_data.HANDataset property*), 51
- processed\_file\_names() (*datasets.matlab\_matrix.MatlabMatrix property*), 52
- processed\_paths() (*data.Dataset property*), 39
- processed\_paths() (*data.dataset.Dataset property*), 35
- ProNE (*class in models.emb.prone*), 65
- ProtainsDataset (*class in datasets.pyg*), 54
- PTCMRDataset (*class in datasets.pyg*), 54
- PTE (*class in models.emb.pte*), 66
- PubMedDataset (*class in datasets.pyg*), 53
- pyg (*in module datasets*), 55
- pyg (*in module models*), 89
- pyg (*in module tasks.unsupervised\_node\_classification*), 45
- ## Q
- QM9Dataset (*class in datasets.pyg*), 54
- ## R
- random\_surfing() (*models.emb.dngr.DNGR method*), 58
- randomly\_choose\_false\_edges() (*in module tasks.link\_prediction*), 43
- raw\_file\_names() (*data.Dataset property*), 39
- raw\_file\_names() (*data.dataset.Dataset property*), 34
- raw\_file\_names() (*datasets.gatne.GatneDataset property*), 47
- raw\_file\_names() (*datasets.gcc\_data.Edgelist property*), 48
- raw\_file\_names() (*datasets.gtn\_data.GTNDataset property*), 49



`raw_file_names()` (*datasets.han\_data.HANDataset* property), 51  
`raw_file_names()` (*datasets.matlab\_matrix.MatlabMatrix* property), 52  
`raw_paths()` (*data.Dataset* property), 39  
`raw_paths()` (*data.dataset.Dataset* property), 35  
`read_gatne_data()` (*in module datasets.gatne*), 47  
`read_gtn_data()` (*datasets.gtn\_data.GTNDataset* method), 49  
`read_gtn_data()` (*datasets.han\_data.HANDataset* method), 51  
`RedditBinary` (*class in datasets.pyg*), 54  
`RedditDataset` (*class in datasets.pyg*), 53  
`RedditMulti12K` (*class in datasets.pyg*), 54  
`RedditMulti5K` (*class in datasets.pyg*), 54  
`register_dataset()` (*in module datasets*), 55  
`register_model()` (*in module models*), 89  
`register_task()` (*in module tasks*), 46  
`regularation()` (*models.emb.hin2vec.Hin2vec\_layer* method), 61  
`reset_parameters()` (*layers.mixhop\_layer.MixHopLayer* method), 27  
`reset_parameters()` (*layers.MixHopLayer* method), 30  
`reset_parameters()` (*models.emb.gatne.GATNEModel* method), 59  
`reset_parameters()` (*models.emb.gatne.NSLoss* method), 59  
`reset_parameters()` (*models.nn.asgcn.ASGCN* method), 68  
`reset_parameters()` (*models.nn.asgcn.GraphConvolution* method), 68  
`reset_parameters()` (*models.nn.fastgcn.GraphConvolution* method), 70  
`reset_parameters()` (*models.nn.gcn.GraphConvolution* method), 72  
`reset_parameters()` (*models.nn.pyg\_diffpool.DiffPool* method), 79  
`reset_parameters()` (*models.nn.pyg\_gtn.GTConv* method), 82  
`reset_parameters()` (*models.nn.pyg\_infograph.InfoGraph* method), 85  
`RowSoftmax` (*class in layers.srgcn\_module*), 29  
`RowUniform` (*class in layers.srgcn\_module*), 29  
`RWGraph` (*class in models.emb.gatne*), 59  
`RWgraph` (*class in models.emb.hin2vec*), 61

**S**

`sample_mask()` (*in module datasets.han\_data*), 50  
`sampler()` (*models.nn.graphsage.Graphsage* method), 73  
`sampling()` (*models.nn.asgcn.ASGCN* method), 68  
`sampling()` (*models.nn.fastgcn.FastGCN* method), 70  
`save_emb()` (*tasks.unsupervised\_graph\_classification.UnsupervisedGraphClassification* method), 45  
`save_emb()` (*tasks.unsupervised\_node\_classification.UnsupervisedNodeClassification* method), 46  
`save_embedding()` (*models.emb.dgk.DeepGraphKernel* method), 57  
`save_embedding()` (*models.emb.graph2vec.Graph2Vec* method), 60  
`scale_matrix()` (*models.emb.dngr.DNGR* method), 58  
`scatter_sum()` (*in module models.nn.pyg\_sortpool*), 86  
`SDNE` (*class in models.emb.sdne*), 67  
`SDNE_layer` (*class in models.emb.sdne*), 67  
`SELayer` (*class in layers*), 30  
`SELayer` (*class in layers.gcc\_module*), 24  
`SELayer` (*class in layers.se\_layer*), 27  
`set_adj()` (*models.nn.asgcn.ASGCN* method), 68  
`set_adj()` (*models.nn.fastgcn.FastGCN* method), 70  
`simulate_walks()` (*models.emb.gatne.RWGraph* method), 59  
`SortPool` (*class in models.nn.pyg\_sortpool*), 86  
`spare2dense_batch()` (*in module models.nn.pyg\_sortpool*), 86  
`SpecialSpm` (*class in models.nn.gat*), 71  
`SpecialSpmFunction` (*class in models.nn.gat*), 71  
`Spectral` (*class in models.emb.spectral*), 67  
`SpGraphAttentionLayer` (*class in models.nn.gat*), 71  
`split_dataset()` (*models.nn.patchy\_san.PatchySAN* class method), 75  
`split_dataset()` (*models.nn.pyg\_dgcnn.DGCNN* class method), 76  
`split_dataset()` (*models.nn.pyg\_diffpool.DiffPool* class method), 78  
`split_dataset()` (*models.nn.pyg\_gin.GIN* class method), 82  
`split_dataset()` (*models.nn.pyg\_infograph.InfoGraph* class method), 84  
`split_dataset()` (*models.nn.pyg\_sortpool.SortPool* class method), 86  
`SRGCN` (*class in models.nn.pyg\_srgcn*), 87  
`SrgcnHead` (*class in models.nn.pyg\_srgcn*), 87  
`SrgcnSoftmaxHead` (*class in models.nn.pyg\_srgcn*), 87  
`sup_forward()` (*models.nn.pyg\_infograph.InfoGraph*

- method*), 85
- sup\_loss() (*models.nn.pyg\_infograph.InfoGraph method*), 85
- SUPEncoder (*class in models.nn.pyg\_infograph*), 84
- SymmetryNorm (*class in layers.srgcn\_module*), 29
- ## T
- task\_name (*in module tasks*), 47
- TASK\_REGISTRY (*in module tasks*), 46
- tasks
- module, 41
  - tasks.base\_task
    - module, 41
  - tasks.graph\_classification
    - module, 41
  - tasks.heterogeneous\_node\_classification
    - module, 42
  - tasks.link\_prediction
    - module, 42
  - tasks.multiplex\_link\_prediction
    - module, 43
  - tasks.multiplex\_node\_classification
    - module, 44
  - tasks.node\_classification
    - module, 44
  - tasks.node\_classification\_sampling
    - module, 44
  - tasks.unsupervised\_graph\_classification
    - module, 45
  - tasks.unsupervised\_node\_classification
    - module, 45
- test\_index() (*datasets.pyg\_modelnet.ModelNetData10 property*), 55
- test\_index() (*datasets.pyg\_modelnet.ModelNetData40 property*), 55
- test\_moco() (*in module models.nn.dgl\_gcc*), 69
- TKipfGCN (*class in models.nn.gcn*), 72
- to() (*data.Data method*), 38
- to() (*data.data.Data method*), 32
- to\_data\_list() (*data.Batch method*), 38
- to\_data\_list() (*data.batch.Batch method*), 30
- to\_list() (*in module data.dataset*), 34
- toBatchedGraph() (*in module models.nn.pyg\_diffpool*), 78
- TopKRanker (*class in tasks.unsupervised\_node\_classification*), 46
- train() (*models.emb.deepwalk.DeepWalk method*), 56
- train() (*models.emb.dngr.DNGR method*), 58
- train() (*models.emb.gatne.GATNE method*), 59
- train() (*models.emb.grarep.GraRep method*), 60
- train() (*models.emb.hin2vec.Hin2vec method*), 61
- train() (*models.emb.hope.HOPE method*), 62
- train() (*models.emb.line.LINE method*), 62
- train() (*models.emb.metapath2vec.Metapath2vec method*), 63
- train() (*models.emb.netmf.NetMF method*), 64
- train() (*models.emb.netsmf.NetSMF method*), 64
- train() (*models.emb.node2vec.Node2vec method*), 65
- train() (*models.emb.prone.ProNE method*), 66
- train() (*models.emb.pte.PTE method*), 66
- train() (*models.emb.sdne.SDNE method*), 67
- train() (*models.emb.spectral.Spectral method*), 68
- train() (*models.nn.dgl\_gcc.GCC method*), 70
- train() (*tasks.base\_task.BaseTask method*), 41
- train() (*tasks.BaseTask method*), 46
- train() (*tasks.graph\_classification.GraphClassification method*), 41
- train() (*tasks.heterogeneous\_node\_classification.HeterogeneousNodeClassification method*), 42
- train() (*tasks.link\_prediction.LinkPrediction method*), 43
- train() (*tasks.multiplex\_link\_prediction.MultiplexLinkPrediction method*), 43
- train() (*tasks.multiplex\_node\_classification.MultiplexNodeClassification method*), 44
- train() (*tasks.node\_classification.NodeClassification method*), 44
- train() (*tasks.node\_classification\_sampling.NodeClassificationSampling method*), 45
- train() (*tasks.unsupervised\_graph\_classification.UnsupervisedGraphClassification method*), 45
- train() (*tasks.unsupervised\_node\_classification.UnsupervisedNodeClassification method*), 46
- train\_index() (*datasets.pyg\_modelnet.ModelNetData10 property*), 55
- train\_index() (*datasets.pyg\_modelnet.ModelNetData40 property*), 55
- TwitterDataset (*class in datasets.gatne*), 48
- ## U
- UNet (*class in models.nn.pyg\_unet*), 88
- uniform\_node\_feature() (*in module tasks.graph\_classification*), 41
- unsup\_forward() (*models.nn.pyg\_infograph.InfoGraph method*), 85
- unsup\_loss() (*models.nn.pyg\_infograph.InfoGraph method*), 85
- unsup\_sup\_loss() (*models.nn.pyg\_infograph.InfoGraph method*), 85
- UnsupervisedGAT (*class in layers.gcc\_module*), 25
- UnsupervisedGIN (*class in layers.gcc\_module*), 25
- UnsupervisedGraphClassification (*class in tasks.unsupervised\_graph\_classification*), 45
- UnsupervisedMPNN (*class in layers.gcc\_module*), 25

UnsupervisedNodeClassification (*class in tasks.unsupervised\_node\_classification*), 46  
untar() (*in module datasets.gtn\_data*), 49  
untar() (*in module datasets.han\_data*), 50  
update() (*layers.maggregator.MeanAggregator method*), 27  
update() (*layers.MeanAggregator method*), 30  
url (*datasets.gatne.GatneDataset attribute*), 47  
url (*datasets.gcc\_data.Edgelist attribute*), 48  
USAAirportDataset (*class in datasets.gcc\_data*), 48  
utils  
    module, 24

## W

walk() (*models.emb.gatne.RWGraph method*), 59  
WikipediaDataset (*class in datasets.matlab\_matrix*), 52  
wl\_iterations() (*models.emb.dgk.DeepGraphKernel static method*), 57  
wl\_iterations() (*models.emb.graph2vec.Graph2Vec static method*), 60

## Y

YouTubeDataset (*class in datasets.gatne*), 48