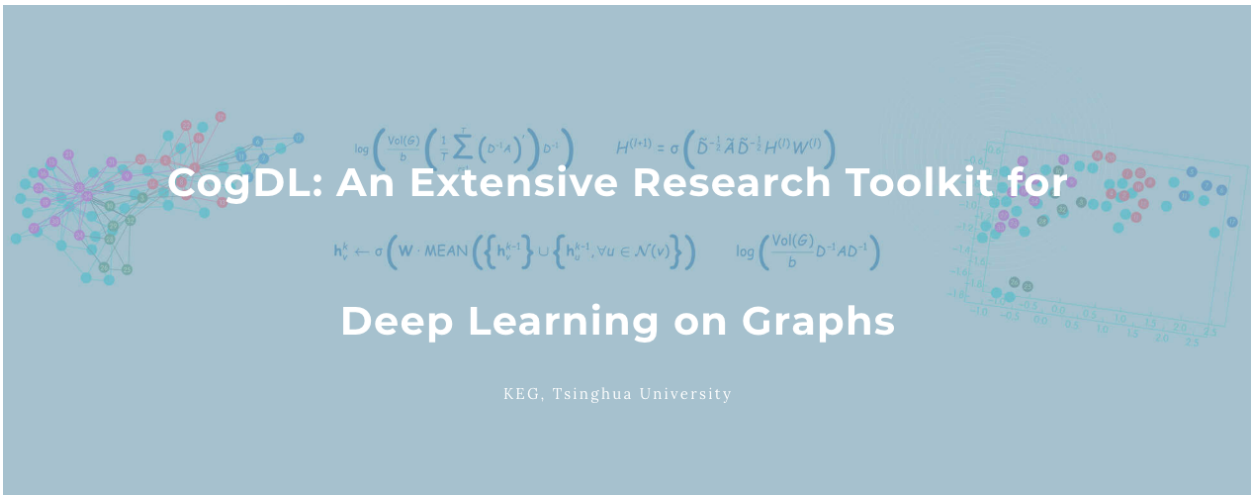
CogDL Documentation

Release 0.5.0b1

KEG

Oct 25, 2021

Get Started



CogDL is a graph representation learning toolkit that allows researchers and developers to easily train and compare baseline or customized models for node classification, graph classification, and other important tasks in the graph domain.

We summarize the contributions of CogDL as follows:

- **High Efficiency**: CogDL utilizes well-optimized operators to speed up training and save GPU memory of GNN models.
- **Easy-to-Use**: CogDL provides easy-to-use APIs for running experiments with the given models and datasets using hyper-parameter search.
- **Extensibility**: The design of CogDL makes it easy to apply GNN models to new scenarios based on our framework.
- **Reproducibility**: CogDL provides reproducible leaderboards for state-of-the-art models on most of important tasks in the graph domain.

CHAPTER 1

News

- The new **v0.5.0b1 pre-release** designs and implements a unified training loop for GNN. It introduces *DataWrapper* to help prepare the training/validation/test data and *ModelWrapper* to define the training/validation/test steps.
- The new **v0.4.1 release** adds the implementation of Deep GNNs and the recommendation task. It also supports new pipelines for generating embeddings and recommendation. Welcome to join our tutorial on KDD 2021 at 10:30 am - 12:00 am, Aug. 14th (Singapore Time). More details can be found in https://kdd2021graph.github.io/.
- The new **v0.4.0 release** refactors the data storage (from `Data` to `Graph`) and provides more fast operators to speed up GNN training. It also includes many self-supervised learning methods on graphs. BTW, we are glad to announce that we will give a tutorial on KDD 2021 in August. Please see this link for more details.
- The new **v0.3.0 release** provides a fast spmm operator to speed up GNN training. We also release the first version of CogDL paper in arXiv. You can join our slack for discussion.
- The new **v0.2.0 release** includes easy-to-use `experiment` and `pipeline` APIs for all experiments and applications. The `experiment` API supports automl features of searching hyper-parameters. This release also provides `OAGBert` API for model inference (`OAGBert` is trained on large-scale academic corpus by our lab). Some features and models are added by the open source community (thanks to all the contributors).
- The new **v0.1.2 release** includes a pre-training task, many examples, OGB datasets, some knowledge graph embedding methods, and some graph neural network models. The coverage of CogDL is increased to 80%. Some new APIs, such as `Trainer` and `Sampler`, are developed and being tested.
- The new **v0.1.1 release** includes the knowledge link prediction task, many state-of-the-art models, and `optuna` support. We also have a Chinese WeChat post about the CogDL release.

CHAPTER 2

Citing CogDL

Please cite our paper if you find our code or results useful for your research:

```
@article{cen2021cogdl,
   title={CogDL: An Extensive Toolkit for Deep Learning on Graphs},
   author={Yukuo Cen and Zhenyu Hou and Yan Wang and Qibin Chen and Yizhen Luo and
↪Xingcheng Yao and Aohan Zeng and Shiguang Guo and Peng Zhang and Guohao Dai and Yu
↪Wang and Chang Zhou and Hongxia Yang and Jie Tang},
   journal={arXiv preprint arXiv:2103.00959},
   year={2021}
}
```

2.1 Install

- Python version >= 3.7
- PyTorch version >= 1.7.1

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

When PyTorch has been installed, cogdl can be installed using pip as follows:

```
pip install cogdl
```

Install from source via:

```
pip install git+https://github.com/thudm/cogdl.git
```

Or clone the repository and install with the following commands:

```
git clone git@github.com:THUDM/cogdl.git
cd cogdl
pip install -e .
```

If you want to use the modules from PyTorch Geometric (PyG), you can follow the instructions to install PyTorch Geometric (https://github.com/rusty1s/pytorch_geometric/#installation).

2.2 Quick Start

2.2.1 API Usage

You can run all kinds of experiments through CogDL APIs, especially `experiment()`. You can also use your own datasets and models for experiments. A quickstart example can be found in the quick_start.py. More examples are provided in the examples/.

```
from cogdl import experiment

# basic usage
experiment(dataset="cora", model="gcn")

# set other hyper-parameters
experiment(dataset="cora", model="gcn", hidden_size=32, max_epoch=200)

# run over multiple models on different seeds
experiment(dataset="cora", model=["gcn", "gat"], seed=[1, 2])

# automl usage
def search_space(trial):
    return {
        "lr": trial.suggest_categorical("lr", [1e-3, 5e-3, 1e-2]),
        "hidden_size": trial.suggest_categorical("hidden_size", [32, 64, 128]),
        "dropout": trial.suggest_uniform("dropout", 0.5, 0.8),
    }

experiment(dataset="cora", model="gcn", seed=[1, 2], search_space=search_space)
```

2.2.2 Command-Line Usage

You can also use `python scripts/train.py --dataset example_dataset --model example_model` to run example_model on example_data.

- `--dataset`, dataset name to run, can be a list of datasets with space like `cora citeseer`. Supported datasets include `cora`, `citeseer`, `pumbed`, `ppi`, `flickr`. More datasets can be found in the cogdl/datasets.
- `--model`, model name to run, can be a list of models like `gcn gat`. Supported models include `gcn`, `gat`, `graphsage`. More models can be found in the cogdl/models.

For example, if you want to run GCN and GAT on the Cora dataset, with 5 different seeds:

```bash python scripts/train.py --dataset cora --model gcn gat --seed 0 1 2 3 4
```

Expected output:

Variant	test_acc	val_acc
('cora', 'gcn')	0.8050±0.0047	0.7940±0.0063
('cora', 'gat')	0.8234±0.0042	0.8088±0.0016

If you want to run parallel experiments on your server with multiple GPUs on multiple models/datasets:

```
python scripts/parallel_train.py --dataset cora citeseer --model gcn gat --devices 0␣
↪1 --seed 0 1 2 3 4
```

Expected output:

Variant	test_acc	val_acc
('cora', 'gcn')	0.8050±0.0047	0.7940±0.0063
('cora', 'gat')	0.8234±0.0042	0.8088±0.0016
('citeseer', 'gcn')	0.6938±0.0133	0.7108±0.0148
('citeseer', 'gat')	0.7098±0.0053	0.7244±0.0039

2.3 Node Classification

Graph neural networks(GNN) have great power in tackling graph-related tasks. In this chapter, we take node classification as an example and show how to use CogDL to finish a workflow using GNN. In supervised setting, node classification aims to predict the ground truth label for each node.

2.3.1 Quick Start

CogDL provides abundant of common benchmark datasets and GNN models. On the one hand, you can simply start a running using models and datasets in CogDL. This is convenient when you want to test the reproducibility of proposed GNN or get baseline results in different datasets.

```
from cogdl import experiment
experiment(model="gcn", dataset="cora", task="node_classification")
```

Or you can create each component separately and manually run the process using `build_dataset`, `build_model`, `build_task` in CogDL.

```
from cogdl.datasets import build_dataset
from cogdl.models import build_model
from cogdl.tasks import build_task

args = build_args_from_dict(dict(task="node_classification", model="gcn", dataset=
↪"cora"))
dataset = build_dataset(args)
model = build_model(args)
task = build_task(args, dataset=dataset, model=model)
task.train()
```

As show above, model/dataset/task are 3 key components in establishing a training process. In fact, CogDL also supports customized model and datasets. This will be introduced in next chapter. In the following we will briefly show the details of each component.

2.3.2 Save trained model

CogDL supports saving the trained model with `save_model` in command line or notebook. For example:

```
experiment(model="gcn", task="node_classification", dataset="cora", save_model="gcn_
↪cora.pt")
```

When the training stops, the model will be saved in *gcn_cora.py*. If you want to continue the training from previous checkpoint with different parameters(such as learning rate, weight decay and etc.), keep the same model parameters (such as hidden size, model layers) and do it as follows:

```
experiment(model="gcn", task="node_classification", dataset="cora", checkpoint="gcn_
↪cora.pt")
```

Or you may just want to do the inference to get prediction results without training. The prediction results will be automatically saved in *gcn_cora.pred*.

```
experiment(model="gcn", task="node_classification", dataset="cora", checkpoint="gcn_
↪cora.pt", inference=True)
```

In command line usage, the same results can be achieved with `--save-model {path}`, `--checkpoint {path}` and `--inference` set.

2.3.3 Save embeddings

Graph representation learning (network embedding and unsupervised GNNs) aims to get node representation. The embeddings can be used in various downstream applications. CogDL will save node embeddings in directory *./embedding*. As shown below, the embeddings will be save in *./embedding/prone_blogcatalog.npy*.

```
experiment(model="prone", dataset="blogcatalog", task="unsupervised_node_
↪classification")
```

Evaluation on node classification will run as the end of training. We follow the same experimental settings used in DeepWalk, Node2Vec and ProNE. We randomly sample different percentages of labeled nodes for training a liblinear classifier and use the remaining for testing We repeat the training for several times and report the average Micro-F1. By default, CogDL samples 90% labeled nodes for training for one time. You are expected to change the setting with `--num-shuffle` and `--training-percents` to your needs.

In addition, CogDL supports evaluating node embeddings without training in different evaluation settings. The following code snippet evaluates the embedding we get above:

```
experiment(
    model="prone",
    dataset="blogcatalog",
    task="unsupervised_node_classification",
    load_emb_path="./embedding/prone_blogcatalog.npy",
    num_shuffle=5,
    training_percents=[0.1, 0.5, 0.9]
)
```

You can also use command line to achieve the same quickly

```
# Get embedding
python script/train.py --model prone --task unsupervised_node_classification --
↪dataset blogcatalog

# Evaluate only
python script/train.py --model prone --task unsupervised_node_classification --
↪dataset blogcatalog --load-emb-path ./embedding/prone_blogcatalog.npy --num-shuffle␣
↪5 --training-percents 0.1 0.5 0.9
```

2.4 Graph Storage

A graph is used to store information of structured data. CogDL represents a graph with a `cogdl.data.Graph` object. Briefly, a `Graph` holds the following attributes:

- `x`: Node feature matrix with shape `[num_nodes, num_features]`, *torch.Tensor*
- `edge_index`: COO format sparse matrix, *Tuple*
- `edge_weight`: Edge weight with shape `[num_edges,]`, *torch.Tensor*
- `edge_attr`: Edge attribute matrix with shape `[num_edges, num_attr]`
- `y`: Target labels of each node, with shape `[num_nodes,]` for single label case and *[num_nodes, num_labels]* for mult-label case
- `row_indptr`: Row index pointer for CSR sparse matrix, *torch.Tensor*.
- `col_indices`: Column indices for CSR sparse matrix, *torch.Tensor*.
- `num_nodes`: The number of nodes in graph.
- `num_edges`: The number of edges in graph.

The above are the basic attributes but are not necessary. You may define a graph with *g = Graph(edge_index=edges)* and omit the others. Besides, `Graph` is not restricted to these attributes and other self-defined attributes, e.x. *graph.mask = mask*, are also supported.

`Graph` stores sparse matrix with COO or CSR format. COO format is easier to add or remove edges, e.x. *add_self_loops*, and CSR is stored for fast message-passing. `Graph` automatically convert between two formats and you can use both on demands without worrying. You can create a Graph with edges or assign edges to a created graph. *edge_weight* will be automatically initialized as all ones, and you can modify it to fit your need.

```
import torch
from cogdl.data import Graph
edges = torch.tensor([[0,1],[1,3],[2,1],[4,2],[0,3]]).t()
g = Graph()
g.edge_index = edges
g = Graph(edge_index=edges) # equivalent to that above
print(g.edge_weight)
>> tensor([1., 1., 1., 1., 1.])
g.num_nodes
>> 5
g.num_edges
>> 5
g.edge_weight = torch.rand(5)
print(g.edge_weight)
>> tensor([0.8399, 0.6341, 0.3028, 0.0602, 0.7190])
```

We also implement commonly used operations in `Graph`:

- `add_self_loops`: add self loops for nodes in graph,

$$\hat{A} = A + I$$

- `add_remaining_self_loops`: add self-loops for nodes without it.
- `sym_norm`: symmetric normalization of edge_weight used *GCN*:

$$\hat{A} = D^{-1/2} A D^{-1/2}$$

- `row_norm`: row-wise normalization of edge_weight:

$$\hat{A} = D^{-1}A$$

- `degrees`: get degrees for each node. For directed graph, this function returns in-degrees of each node.

```
import torch
from cogdl.data import Graph
edge_index = torch.tensor([[0,1],[1,3],[2,1],[4,2],[0,3]]).t()
g = Graph(edge_index=edge_index)
>> Graph(edge_index=[2, 5])
g.add_remaining_self_loops()
>> Graph(edge_index=[2, 10], edge_weight=[10])
>> print(edge_weight) # tensor([1., 1., ..., 1.])
g.row_norm()
>> print(edge_weight) # tensor([0.3333, ..., 0.50])
```

- `subgraph`: get a subgraph containing given nodes and edges between them.
- `edge_subgraph`: get a subgraph containing given edges and corresponding nodes.
- `sample_adj`: sample a fixed number of neighbors for each given node.

```
from cogdl.datasets import build_dataset_from_name
g = build_dataset_from_name("cora")[0]
g.num_nodes
>> 2707
g.num_edges
>> 10184
# Get a subgraph contaning nodes [0, .., 99]
sub_g = g.subgraph(torch.arange(100))
>> Graph(x=[100, 1433], edge_index=[2, 18], y=[100])
# Sample 3 neighbors for each nodes in [0, .., 99]
nodes, adj_g = g.sample_adj(torch.arange(100), size=3)
>> Graph(edge_index=[2, 300]) # adj_g
```

- `train/eval`: In inductive settings, some nodes and edges are unseen during training, `train/eval` provides access to switching backend graph for training/evaluation. In transductive setting, you may ignore this.

```
# train_step
model.train()
graph.train()

# inference_step
model.eval()
data.eval()
```

2.4.1 Mini-batch Graphs

In node classification, all operations are in one single graph. But in tasks like graph classification, we need to deal with many graphs with mini-batch. Datasets for graph classification contains graphs which can be accessed with index, e.x. `data[2]`. To support mini-batch training/inference, CogDL combines graphs in a batch into one whole graph, where adjacency matrices form sparse block diagnal matrices and others(node features, labels) are concatenated in node dimension. `cogdl.data.Dataloader` handles the process.

```
from cogdl.data import DataLoader
from cogdl.datasets import build_dataset_from_name

dataset = build_dataset_from_name("mutag")
```

(continues on next page)

(continued from previous page)

```
>> MUTAGDataset(188)
dataswet[0]
>> Graph(x=[17, 7], y=[1], edge_index=[2, 38])
loader = DataLoader(dataset, batch_size=8)
for batch in loader:
    model(batch)
>> Batch(x=[154, 7], y=[8], batch=[154], edge_index=[2, 338])
```

`batch` is an additional attributes that indicate the respective graph the node belongs to. It is mainly used to do global pooling, or called *readout* to generate graph-level representation. Concretely, `batch` is a tensor like:

$$batch = [0, .., 0, 1, ..., 1, N-1, ..., N-1]$$

The following code snippet shows how to do global pooling to sum over features of nodes in each graph:

```
def batch_sum_pooling(x, batch):
    batch_size = int(torch.max(batch.cpu())) + 1
    res = torch.zeros(batch_size, x.size(1)).to(x.device)
    out = res.scatter_add_(
        dim=0,
        index=batch.unsqueeze(-1).expand_as(x),
        src=x
       )
    return out
```

2.4.2 Editing Graphs

Mutation or changes can be applied to edges in some settings. In such cases, we need to *generate* a graph for calculation while keep the original graph. CogDL provides *graph.local_graph* to set up a local scape and any out-of-place operation will not reflect to the original graph. However, in-place operation will affect the original graph.

```
graph = build_dataset_from_name("cora")[0]
graph.num_edges
>> 10184
with graph.local_graph():
    mask = torch.arange(100)
    row, col = graph.edge_index
    graph.edge_index = (row[mask], col[mask])
    graph.num_edges
    >> 100
graph.num_edges
>> 10184

graph.edge_weight
>> tensor([1.,...,1.])
with graph.local_graph():
    graph.edge_weight += 1
graph.edge_weight
>> tensor([2.,...,2.])
```

2.4.3 Common benchmarks

CogDL provides a bunch of commonly used datasets for graph tasks like node classification, graph classification and many others. You can access them conveniently shown as follows. Statistics of datasets are on this page .

```
from cogdl.datasets import build_dataset_from_name, build_dataset
dataset = build_dataset_from_name("cora")
dataset = build_dataset(args) # args.dataet = "cora"
```

For all datasets for node classification, we use *train_mask*, *val_mask*, *test_mask* to denote train/validation/test split for nodes.

2.5 Using customized GNN

Sometimes you would like to design your own GNN module or use GNN for other purposes. In this chapter, we introduce how to use GNN layer in CogDL to write your own GNN model and how to write a GNN layer from scratch.

2.5.1 GNN layers in CogDL to Define model

CogDL has implemented popular GNN layers in `cogdl.layers`, and they can serve as modules to help design new GNNs. Here is how we implement Jumping Knowledge Network (JKNet) with `GCNLayer` in CogDL.

JKNet collects the output of all layers and concatenate them together to get the result:

$$H^{(0)} = X$$
$$H^{(i+1)} = \sigma(\hat{A}H^{(i)}W^{(i)}$$
$$OUT = CONCAT([H^{(0)}, ..., H^{(L)})$$

```
import torch
from cogdl.models import BaseModel

class JKNet(BaseModel):
    def __init__(self, in_feats, out_feats, hidden_size, num_layers):
        super(JKNet, self).__init__()
        shapes = [in_feats] + [hidden_size] * num_layers
        #
        self.layers = nn.ModuleList([
            GCNLayer(shape[i], shape[i+1])
            for i in range(num_layers)
        ])
        self.fc = nn.Linear(hidden_size * num_layers, out_feats)

    def forward(self, graph):
        graph.add_remaining_self_loops()
        graph.sym_norm()
        h = graph.x
        out = []
        for layer in self.layers:
            h = layer(x)
            out.append(h)
        out = torch.cat(out, dim=1)
        return self.fc(out)
```

2.5.2 Define your GNN Module

In most cases, you may build a layer module with new message propagation and aggragation scheme. Here the code snippet shows how to implement a GCNLayer using `Graph` and efficient sparse matrix operators in CogDL.

```
import torch
from cogdl.utils import spmm

class GCNLayer(torch.nn.Module):
    """
    Args:
        in_feats: int
            Input feature size
        out_feats: int
            Output feature size
    """
    def __init__(self, in_feats, out_feats):
        super(GCNLayer, self).__init__()
        self.fc = torch.nn.Linear(in_feats, out_feats)

    def forward(self, graph, x):
        # symmetric normalization of adjacency matrix
        graph.sym_norm()
        h = self.fc(x)
        h = spmm(graph, h)
        return h
```

`spmm` is sparse matrix multiplication operation frequently used in GNNs.

$$H = AH = SpMM(A, H)$$

Sparse matrix is stored in `Graph` and will be called automatically. Message-passing in spatial space is equivalent to matrix operations. CogDL also supports other efficient operators like `edge_softmax` and `multi_head_spmm`, you can refer to this page for usage.

2.5.3 Use Custom models with CogDL

Now that you have defined your own GNN, you can use dataset/task in CogDL to immediately train and evaluate the performance of your model.

```
data = dataset.data
# Use the JKNet model as defined above
model = JKNet(data.num_features, data.num_classes, 32, 4)
experiment(model=model, dataset="cora", mw="node_classification_mw", dw="node_
↪classification_dw")
```

2.6 Using customized Dataset

CogDL has provided lots of common datasets. But you may wish to apply GNN to new datasets for different applications. CogDL provides an interface for customized datasets. You take care of reading in the dataset and the rest is to CogDL

We provide `NodeDataset` and `GraphDataset` as abstract classes and implement necessary basic operations.

2.6.1 Dataset for node_classification

To create a dataset for node_classification, you need to inherit `NodeDataset`. `NodeDataset` is for tasks like *node_classification* or *unsupervised_node_classification*, which focus on node-level prediction. Then you need to implement `process` method. In this method, you are expected to read in your data and preprocess raw data to the format available to CogDL with `Graph`. Afterwards, we suggest you to save the processed data (we will also help you do it as you return the data) to avoid doing the preprocessing again. Next time you run the code, CogDL will directly load it.

The running process of the module is as follows:

1. Specify the path to save processed data with *self.path*

2. Function *process* is called to load and preprocess data and your data is saved as *Graph* in *self.path*. This step will be implemented the first time you use your dataset. And then every time you use your dataset, the dataset will be loaded from *self.path* for convenience. 3. For dataset, for example, named *MyNodeDataset* in node-level tasks, You can access the data/Graph via *MyNodeDataset.data* or *MyDataset[0]*.

In addition, evaluation metric for your dataset should be specified. CogDL provides `accuracy` and `multiclass_f1` for multi-class classification, `multilabel_f1` for multi-label classification.

If `scale_feat` is set to be *True*, CogDL will normalize node features with mean u and variance s:

$$z = (x - u)/s$$

Here is an example:

```
from cogdl.data import Graph
from cogdl.datasets import NodeDataset, register_dataset

@register_dataset("node_dataset")
class MyNodeDataset(NodeDataset):
    def __init__(self, path="data.pt"):
        self.path = path
        super(MyNodeDataset, self).__init__(path, scale_feat=False, metric="accuracy")

    def process(self):
        """You need to load your dataset and transform to `Graph`"""
        # Load and preprocess data
        edge_index = torch.tensor([[0, 1], [0, 2], [1, 2], [1, 3]).t()
        x = torch.randn(4, 10)
        mask = torch.bool(4)
        # Provide attributes as you need and save the data into `Graph`
        data = Graph(x=x, edge_index=edge_index)
        torch.save(data, self.path)
        return data

dataset = MyNodeDataset("data.pt")
```

2.6.2 Dataset for graph_classification

Similarly, you need to inherit `GraphDataset` when you want to build a dataset for graph-level tasks such as *graph_classification*. The overall implementation is similar while the difference is in `process`. As `GraphDataset` contains a lot of graphs, you need to transform your data to `Graph` for each graph separately to form a list of `Graph`. An example is shown as follows:

```
from cogdl.datasets import GraphDataset

@register_dataset("graph_dataset")
class MyGraphDataset(GraphDataset):
    def __init__(self, path="data.pt"):
        self.path = path
        super(MyGraphDataset, self).__init__(path, metric="accuracy")

    def process(self):
        # Load and preprocess data
        # Here we randomly generate several graphs for simplicity as an example
        graphs = []
        for i in range(10):
            edges = torch.randint(0, 20, (2, 30))
            label = torch.randint(0, 7, (1,))
            graphs.append(Graph(edge_index=edges, y=label))
        torch.save(graphs, self.path)
        return graphs
```

2.6.3 Use custom dataset with CogDL

Now that you have set up your dataset, you can use models/task in CogDL immediately to get results.

```
# Use the GCN model with the dataset we define above
dataset = MyNodeDataset("data.pt")
args.model = "gcn"
task = build_task(args, dataset=dataset)
task.train()

# Or you may simple run the command after `register_dataset`
experiment(model="gcn", task="node_classification", dataset="node_dataset")

# That's the same for other tasks
experiment(model="gin", task="graph_classification", dataset="graph_dataset")
```

2.7 data

class cogdl.data.**Graph**(*x=None*, *y=None*, ***kwargs*)

Bases: cogdl.data.data.BaseGraph

add_remaining_self_loops()

clone()

col_indices

col_norm()

csr_subgraph(*node_idx*, *keep_order=False*)

degrees()

device

edge_attr

edge_index

edge_subgraph(*edge_idx*, *require_idx=True*)

edge_types

edge_weight
Return actual edge_weight

eval()

static from_dict(*dictionary*)
Creates a data object from a python dictionary.

static from_pyg_data(*data*)

in_norm

is_inductive()

is_symmetric()

keys
Returns all names of graph attributes.

local_graph()

mask2nid(*split*)

nodes()

normalize(*key='sym'*)

num_classes

num_edges
Returns the number of edges in the graph.

num_features
Returns the number of features per node in the graph.

num_nodes

out_norm

padding_self_loops()

random_walk(*seeds*, *max_nodes_per_seed*, *restart_p=0.0*)

random_walk_with_restart(*seeds*, *max_nodes_per_seed*, *restart_p=0.0*)

raw_edge_weight
Return edge_weight without __in_norm__ and __out_norm__, only used for SpMM

remove_self_loops()

restore(*key*)

row_indptr

row_norm()

sample_adj(*batch*, *size=-1*, *replace=True*)

set_asymmetric()

set_symmetric()

store(*key*)

subgraph(*node_idx*, *keep_order=False*)

sym_norm()

test_nid

to_networkx()

to_scipy_csr()

train()

train_nid

val_nid

class cogdl.data.**Adjacency**(*row=None*, *col=None*, *row_ptr=None*, *weight=None*, *attr=None*, *num_nodes=None*, *types=None*, ***kwargs*)

Bases: cogdl.data.data.BaseGraph

add_remaining_self_loops()

clone()

col_norm()

convert_csr()

degrees(*node_idx=None*)

device

edge_index

static from_dict(*dictionary*)

Creates a data object from a python dictionary.

generate_normalization(*norm='sym'*)

get_weight(*indicator=None*)

If *indicator* is not None, the normalization will not be implemented

is_symmetric()

keys

Returns all names of graph attributes.

normalize_adj(*norm='sym'*)

num_edges

num_nodes

padding_self_loops()

random_walk(*seeds*, *length=1*, *restart_p=0.0*)

remove_self_loops()

row_indptr

row_norm()

row_ptr_v

set_symmetric(*val*)

set_weight(*weight*)

sym_norm()

to_networkx(*weighted=True*)

to_scipy_csr()

class cogdl.data.**Batch**(*batch=None*, ***kwargs*)
Bases: cogdl.data.data.Graph

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.

cumsum(*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.

---

static from_data_list(*data_list*, *class_type=None*)
Constructs a batch object from a python list holding cogdl.data.Data objects. The assignment vector batch is created on the fly. Additionally, creates assignment batch vectors for each key in follow_batch.

num_graphs
Returns the number of graphs in the batch.

class cogdl.data.**Dataset**(*root*, *transform=None*, *pre_transform=None*, *pre_filter=None*)
Bases: torch.utils.data.dataset.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)

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

download()
Downloads the dataset to the self.raw_dir folder.

edge_attr_size

get(*idx*)
Gets the data object at index idx.

get_evaluator()

get_loss_fn()

max_degree

max_graph_size

num_classes
The number of classes in the dataset.

num_features
Returns the number of features per node in the graph.

num_graphs

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

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

raw_paths
The filepaths to find in order to skip the download.

class cogdl.data.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)
Bases: `torch.utils.data.dataloader.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`)

static collate_fn(*batch*)

get_parameters()

record_parameters(*params*)

class cogdl.data.**MultiGraphDataset**(*root=None*, *transform=None*, *pre_transform=None*, *pre_filter=None*)
Bases: `cogdl.data.dataset.Dataset`

get(*idx*)
Gets the data object at index `idx`.

len()

max_degree

max_graph_size

num_classes
The number of classes in the dataset.

num_features
Returns the number of features per node in the graph.

num_graphs

cogdl.data.**batch_graphs**(*graphs*)

2.8 datasets

2.8.1 GATNE dataset

class cogdl.datasets.gatne.**AmazonDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gatne.GatneDataset*

class cogdl.datasets.gatne.**GatneDataset**(*root*, *name*)
Bases: cogdl.data.dataset.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").

download()
Downloads the dataset to the self.raw_dir folder.

get(*idx*)
Gets the data object at index idx.

process()
Processes the dataset to the self.processed_dir folder.

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

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

url = 'https://github.com/THUDM/GATNE/raw/master/data'

class cogdl.datasets.gatne.**TwitterDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gatne.GatneDataset*

class cogdl.datasets.gatne.**YouTubeDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gatne.GatneDataset*

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

2.8.2 GCC dataset

class cogdl.datasets.gcc_data.**Edgelist**(*root*, *name*)
Bases: cogdl.data.dataset.Dataset

download()
Downloads the dataset to the self.raw_dir folder.

get(*idx*)
Gets the data object at index idx.

num_classes
The number of classes in the dataset.

process()
Processes the dataset to the self.processed_dir folder.

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

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

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

class cogdl.datasets.gcc_data.**GCCDataset**(*root*, *name*)
Bases: cogdl.data.dataset.Dataset

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

preprocess(*root*, *name*)

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

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

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

class cogdl.datasets.gcc_data.**KDD_ICDM_GCCDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gcc_data.GCCDataset*

class cogdl.datasets.gcc_data.**SIGIR_CIKM_GCCDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gcc_data.GCCDataset*

class cogdl.datasets.gcc_data.**SIGMOD_ICDE_GCCDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gcc_data.GCCDataset*

class cogdl.datasets.gcc_data.**USAAirportDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gcc_data.Edgelist*

2.8.3 GTN dataset

class cogdl.datasets.gtn_data.**ACM_GTNDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gtn_data.GTNDataset*

class cogdl.datasets.gtn_data.**DBLP_GTNDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gtn_data.GTNDataset*

class cogdl.datasets.gtn_data.**GTNDataset**(*root*, *name*)
Bases: cogdl.data.dataset.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"`).

apply_to_device(*device*)

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

num_classes
The number of classes in the dataset.

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

read_gtn_data(*folder*)

class cogdl.datasets.gtn_data.**IMDB_GTNDataset**(*data_path='data'*)
Bases: *cogdl.datasets.gtn_data.GTNDataset*

2.8.4 HAN dataset

class cogdl.datasets.han_data.**ACM_HANDataset**(*data_path='data'*)
Bases: *cogdl.datasets.han_data.HANDataset*

class cogdl.datasets.han_data.**DBLP_HANDataset**(*data_path='data'*)
Bases: *cogdl.datasets.han_data.HANDataset*

class cogdl.datasets.han_data.**HANDataset**(*root*, *name*)
Bases: cogdl.data.dataset.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"`).

apply_to_device(*device*)

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

num_classes
The number of classes in the dataset.

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

read_gtn_data(*folder*)

class cogdl.datasets.han_data.**IMDB_HANDataset**(*data_path='data'*)
Bases: *cogdl.datasets.han_data.HANDataset*

cogdl.datasets.han_data.**sample_mask**(*idx*, *length*)
Create mask.

2.8.5 KG dataset

class cogdl.datasets.kg_data.**FB13Datset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

class cogdl.datasets.kg_data.**FB13SDatset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

class cogdl.datasets.kg_data.**FB15k237Datset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

class cogdl.datasets.kg_data.**FB15kDatset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

class cogdl.datasets.kg_data.**KnowledgeGraphDataset**(*root*, *name*)
Bases: cogdl.data.dataset.Dataset

download()
Downloads the dataset to the self.raw_dir folder.

get(*idx*)
Gets the data object at index idx.

num_entities

num_relations

process()
Processes the dataset to the self.processed_dir folder.

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

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

test_start_idx

train_start_idx

url = 'https://cloud.tsinghua.edu.cn/d/b567292338f2488699b7/files/?p=%2F{}%2F{}&dl=1'

valid_start_idx

class cogdl.datasets.kg_data.**WN18Datset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

class cogdl.datasets.kg_data.**WN18RRDataset**(*data_path='data'*)
Bases: *cogdl.datasets.kg_data.KnowledgeGraphDataset*

cogdl.datasets.kg_data.**read_triplet_data**(*folder*)

2.8.6 Matlab matrix dataset

class cogdl.datasets.matlab_matrix.**BlogcatalogDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.MatlabMatrix*

class cogdl.datasets.matlab_matrix.**DblpNEDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.NetworkEmbeddingCMTYDataset*

class cogdl.datasets.matlab_matrix.**FlickrDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.MatlabMatrix*

class cogdl.datasets.matlab_matrix.**MatlabMatrix**(*root*, *name*, *url*)
Bases: cogdl.data.dataset.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"`).

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

num_classes
The number of classes in the dataset.

num_nodes

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

class cogdl.datasets.matlab_matrix.**NetworkEmbeddingCMTYDataset**(*root*, *name*, *url*)
Bases: cogdl.data.dataset.Dataset

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

num_classes
The number of classes in the dataset.

num_nodes

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

class cogdl.datasets.matlab_matrix.**PPIDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.MatlabMatrix*

class cogdl.datasets.matlab_matrix.**WikipediaDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.MatlabMatrix*

class cogdl.datasets.matlab_matrix.**YoutubeNEDataset**(*data_path='data'*)
Bases: *cogdl.datasets.matlab_matrix.NetworkEmbeddingCMTYDataset*

2.8.7 PyG OGB dataset

class cogdl.datasets.ogb.**OGBArxivDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBNDataset*

get_evaluator()

class cogdl.datasets.ogb.**OGBCodeDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBGDataset*

class cogdl.datasets.ogb.**OGBGDataset**(*root*, *name*)
Bases: cogdl.data.dataset.Dataset

get(*idx*)
Gets the data object at index idx.

get_loader(*args*)

get_subset(*subset*)

num_classes
The number of classes in the dataset.

class cogdl.datasets.ogb.**OGBMolbaceDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBGDataset*

class cogdl.datasets.ogb.**OGBMolhivDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBGDataset*

class cogdl.datasets.ogb.**OGBMolpcbaDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBGDataset*

class cogdl.datasets.ogb.**OGBNDataset**(*root*, *name*, *transform=None*)
Bases: cogdl.data.dataset.Dataset

get(*idx*)
Gets the data object at index idx.

get_evaluator()

get_loss_fn()

process()
Processes the dataset to the self.processed_dir folder.

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

class cogdl.datasets.ogb.**OGBPapers100MDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBNDataset*

class cogdl.datasets.ogb.**OGBPpaDataset**
Bases: *cogdl.datasets.ogb.OGBGDataset*

class cogdl.datasets.ogb.**OGBProductsDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBNDataset*

class cogdl.datasets.ogb.**OGBProteinsDataset**(*data_path='data'*)
Bases: *cogdl.datasets.ogb.OGBNDataset*

edge_attr_size

get_evaluator()

get_loss_fn()

process()
Processes the dataset to the `self.processed_dir` folder.

2.8.8 TU dataset

class cogdl.datasets.tu_data.**CollabDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**ENZYMES**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**ImdbBinaryDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**ImdbMultiDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**MUTAGDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**NCI109Dataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**NCI1Dataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**PTCMRDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**ProteinsDataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**RedditBinary**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**RedditMulti12K**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**RedditMulti5K**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**TUDataset**(*root*, *name*)
Bases: cogdl.data.dataset.MultiGraphDataset

download()
Downloads the dataset to the `self.raw_dir` folder.

num_classes
The number of classes in the dataset.

process()
Processes the dataset to the `self.processed_dir` folder.

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

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

url = 'https://www.chrsmrrs.com/graphkerneldatasets'

cogdl.datasets.tu_data.**cat**(*seq*)

cogdl.datasets.tu_data.**coalesce**(*index*, *value*, *m*, *n*)

cogdl.datasets.tu_data.**normalize_feature**(*data*)

cogdl.datasets.tu_data.**num_edge_attributes**(*edge_attr=None*)

cogdl.datasets.tu_data.**num_edge_labels**(*edge_attr=None*)

cogdl.datasets.tu_data.**num_node_attributes**(*x=None*)

cogdl.datasets.tu_data.**num_node_labels**(*x=None*)

cogdl.datasets.tu_data.**parse_txt_array**(*src*, *sep=None*, *start=0*, *end=None*, *dtype=None*, *device=None*)

cogdl.datasets.tu_data.**read_file**(*folder*, *prefix*, *name*, *dtype=None*)

cogdl.datasets.tu_data.**read_tu_data**(*folder*, *prefix*)

cogdl.datasets.tu_data.**read_txt_array**(*path*, *sep=None*, *start=0*, *end=None*, *dtype=None*, *device=None*)

cogdl.datasets.tu_data.**segment**(*src*, *indptr*)

2.8.9 Module contents

cogdl.datasets.**build_dataset**(*args*)

cogdl.datasets.**build_dataset_from_name**(*dataset*)

cogdl.datasets.**build_dataset_from_path**(*data_path*, *dataset=None*)

cogdl.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

cogdl.datasets.**try_adding_dataset_args**(*dataset*, *parser*)

2.9 models

2.9.1 BaseModel

class cogdl.models.base_model.**BaseModel**

Bases: torch.nn.modules.module.Module

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

Build a new model instance.

device

forward(**args*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

set_loss_fn(*loss_fn*)

2.9.2 Embedding Model

class cogdl.models.emb.hope.**HOPE**(*dimension*, *beta*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
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

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.spectral.**Spectral**(*hidden_size*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

train(*graph*, *return_dict=False*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.hin2vec.**Hin2vec**(*hidden_dim*, *walk_length*, *walk_num*, *batch_size*, *hop*, *negative*, *epochs*, *lr*, *cpu=True*)

Bases: *cogdl.models.base_model.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 nagative samples for each meta2path pair. epochs (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(*args*)

Build a new model instance.

forward(*data*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

train(*data*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

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

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.deepwalk.**DeepWalk**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *iteration*)
Bases: *cogdl.models.base_model.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: argparse.ArgumentParser*)
Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*) → cogdl.models.emb.deepwalk.DeepWalk
Build a new model instance.

forward(*graph*, *embedding_model_creator=<class 'gensim.models.word2vec.Word2Vec'>*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

---

train(*graph*, *embedding_model_creator=<class 'gensim.models.word2vec.Word2Vec'>*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.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: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*network_data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*network_data*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.dgk.**DeepGraphKernel**(*hidden_dim*, *min_count*, *window_size*, *sampling_rate*, *rounds*, *epoch*, *alpha*, *n_workers=4*)

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

static feature_extractor(*data*, *rounds*, *name*)

forward(*graphs*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

save_embedding(*output_path*)

static wl_iterations(*graph*, *features*, *rounds*)

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

Bases: *cogdl.models.base_model.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 transitition probability.

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

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

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.dngr.**DNGR**(*hidden_size1*, *hidden_size2*, *noise*, *alpha*, *step*, *max_epoch*, *lr*, *cpu*)

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_denoised_matrix(*mat*)

get_emb(*matrix*)

get_ppmi_matrix(*mat*)

random_surfing(*adj_matrix*)

scale_matrix(*mat*)

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.pronepp.**ProNEPP**(*filter_types*, *svd*, *search*, *max_evals=None*, *loss_type=None*, *n_workers=None*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

class cogdl.models.emb.graph2vec.**Graph2Vec**(*dimension*, *min_count*, *window_size*, *dm*, *sampling_rate*, *rounds*, *epoch*, *lr*, *worker=4*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

static feature_extractor(*data*, *rounds*, *name*)

forward(*graphs*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

save_embedding(*output_path*)

static wl_iterations(*graph*, *features*, *rounds*)

class cogdl.models.emb.metapath2vec.**Metapath2vec**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *iteration*, *schema*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

train(*data*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.node2vec.**Node2vec**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *iteration*, *p*, *q*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.pte.**PTE**(*dimension*, *walk_length*, *walk_num*, *negative*, *batch_size*, *alpha*)
Bases: *cogdl.models.base_model.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 nagative 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(*args*)
Build a new model instance.

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*data*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.netsmf.**NetSMF**(*dimension*, *window_size*, *negative*, *num_round*, *worker*)

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.line.**LINE**(*dimension*, *walk_length*, *walk_num*, *negative*, *batch_size*, *alpha*, *order*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.sdne.**SDNE**(*hidden_size1*, *hidden_size2*, *droput*, *alpha*, *beta*, *nu1*, *nu2*, *max_epoch*, *lr*, *cpu*)
Bases: *cogdl.models.base_model.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. droput (float) : Droput 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(*args*)
Build a new model instance.

forward(*graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

class cogdl.models.emb.prone.**ProNE**(*dimension*, *step*, *mu*, *theta*)
Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*graph: cogdl.data.data.Graph*, *return_dict=False*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*graph*, *return_dict=False*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

2.9.3 GNN Model

class cogdl.models.nn.dgi.**DGIModel**(*in_feats*, *hidden_size*, *activation*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

embed(*data*)

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.models.nn.mvgrl.**MVGRL**(*in_feats*, *hidden_size*, *sample_size=2000*, *batch_size=4*, *alpha=0.2*, *dataset='cora'*)
Bases: *cogdl.models.base_model.BaseModel*

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

augment(*graph*)

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

embed(*data*, *msk=None*)

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

loss(*data*)

preprocess(*graph*)

class cogdl.models.nn.patchy_san.**PatchySAN**(*num_features*, *num_classes*, *num_sample*, *num_neighbor*, *iteration*)

Bases: *cogdl.models.base_model.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.

build_model(*num_channel*, *num_sample*, *num_neighbor*, *num_class*)

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

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.pyg_cheb.**Chebyshev**(*in_feats*, *hidden_size*, *out_feats*, *num_layers*, *dropout*, *filter_size*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.gcn.**GCN**(*in_feats*, *hidden_size*, *out_feats*, *num_layers*, *dropout*, *activation='relu'*, *residual=False*, *norm=None*, *actnn=False*, *rp_ratio=1*)

Bases: *cogdl.models.base_model.BaseModel*

The GCN model from the "Semi-Supervised Classification with Graph Convolutional Networks" paper

Args: in_features (int) : Number of input features. out_features (int) : Number of classes. hidden_size (int) : The dimension of node representation. dropout (float) : Dropout rate for model training.

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

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

embed(*graph*)

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.gdc_gcn.**GDC_GCN**(*nfeat*, *nhid*, *nclass*, *dropout*, *alpha*, *t*, *k*, *eps*, *gdctype*)
Bases: *cogdl.models.base_model.BaseModel*

The GDC model from the "Diffusion Improves Graph Learning" paper, with the PPR and heat matrix variants combined with GCN

Args: num_features (int) : Number of input features in ppr-preprocessed dataset. num_classes (int) : Number of classes. hidden_size (int) : The dimension of node representation. dropout (float) : Dropout rate for model training. alpha (float) : PPR polynomial filter param, 0 to 1. t (float) : Heat polynomial filter param k (int) : Top k nodes retained during sparsification. eps (float) : Threshold for clipping. gdc_type (str) : "none", "ppr", "heat"

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data=None*)

preprocessing(*data*, *gdc_type='ppr'*)

reset_data(*data*)

class cogdl.models.nn.graphsage.**Graphsage**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *sample_size*, *dropout*, *aggr*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(**args*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

inference(*x_all*, *data_loader*)

mini_forward(*graph*)

sampling(*edge_index*, *num_sample*)

class cogdl.models.nn.compgcn.**LinkPredictCompGCN**(*num_entities*, *num_rels*, *hidden_size*, *num_bases=0*, *layers=1*, *sampling_rate=0.01*, *penalty=0.001*, *dropout=0.0*, *lbl_smooth=0.1*, *opn='sub'*)
Bases: *cogdl.utils.link_prediction_utils.GNNLinkPredict*, *cogdl.models.base_model.BaseModel*

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

add_reverse_edges(*edge_index*, *edge_types*)

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

loss(*data: cogdl.data.data.Graph*, *scoring*)

predict(*graph*)

class cogdl.models.nn.drgcn.**DrGCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*, *norm=None*, *activation='relu'*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*graph*)

class cogdl.models.nn.pyg_graph_unet.**GraphUnet**(*in_feats: int, hidden_size: int, out_feats: int, pooling_layer: int, pooling_rates: List[float], n_dropout: float = 0.5, adj_dropout: float = 0.3, activation: str = 'elu', improved: bool = False, aug_adj: bool = False*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph: cogdl.data.data.Graph*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.models.nn.gcnmix.**GCNMix**(*in_feat*, *hidden_size*, *num_classes*, *k*, *temperature*, *alpha*, *dropout*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

forward_aux(*x*, *label*, *train_index*, *mix_hidden=True*, *layer_mix=1*)

predict_noise(*data*, *tau=1*)

class cogdl.models.nn.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: *cogdl.models.base_model.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 embeded 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 poolling 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.

after_pooling_forward(*gnn_layers*, *adj*, *x*, *concat=False*)

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

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

graph_classificatoin_loss(*batch*)

reset_parameters()

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.gcnii.**GCNII**(*in_feats*, *hidden_size*, *out_feats*, *num_layers*, *dropout=0.5*, *alpha=0.1*, *lmbda=1*, *wd1=0.0*, *wd2=0.0*, *residual=False*, *actnn=False*)

Bases: *cogdl.models.base_model.BaseModel*

Implementation of GCNII in paper *"Simple and Deep Graph Convolutional Networks"* <https://arxiv.org/abs/2007.02133>.

in_feats [int] Size of each input sample

hidden_size [int] Size of each hidden unit

out_feats [int] Size of each out sample

num_layers : int dropout : float alpha : float

Parameter of initial residual connection

lmbda [float] Parameter of identity mapping

wd1 [float] Weight-decay for Fully-connected layers

wd2 [float] Weight-decay for convolutional layers

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_optimizer(*args*)

predict(*graph*)

class cogdl.models.nn.sign.**MLP**(*in_feats*, *out_feats*, *hidden_size*, *num_layers*, *dropout=0.0*, *activation='relu'*, *norm=None*, *act_first=False*, *bias=True*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.pyg_gcn.**GCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_embeddings(*x*, *edge_index*, *weight=None*)

class cogdl.models.nn.mixhop.**MixHop**(*num_features*, *num_classes*, *dropout*, *layer1_pows*, *layer2_pows*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.gat.**GAT**(*in_feats*, *hidden_size*, *out_features*, *num_layers*, *dropout*, *attn_drop*, *alpha*, *nhead*, *residual*, *last_nhead*, *norm=None*)
Bases: *cogdl.models.base_model.BaseModel*

The GAT model from the "Graph Attention Networks" paper

Args: num_features (int) : Number of input features. num_classes (int) : Number of classes. hidden_size (int) : The dimension of node representation. dropout (float) : Dropout rate for model training. alpha (float) : Coefficient of leaky_relu. nheads (int) : Number of attention heads.

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

predict(*graph*)

class cogdl.models.nn.han.**HAN**(*num_edge*, *w_in*, *w_out*, *num_class*, *num_nodes*, *num_layers*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.models.nn.ppnp.**PPNP**(*nfeat*, *nhid*, *nclass*, *num_layers*, *dropout*, *propagation*, *alpha*, *niter*, *cache=True*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

predict(*graph*)

class cogdl.models.nn.grace.**GRACE**(*in_feats: int, hidden_size: int, proj_hidden_size: int, num_layers: int, drop_feature_rates: List[float], drop_edge_rates: List[float], tau: float = 0.5, activation: str = 'relu', batch_size: int = -1*)
Bases: *cogdl.models.base_model.BaseModel*

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

augment(*graph*)

batched_loss(*z1: torch.Tensor*, *z2: torch.Tensor*, *batch_size: int*)

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

contrastive_loss(*z1: torch.Tensor*, *z2: torch.Tensor*)

drop_adj(*graph: cogdl.data.data.Graph*, *drop_rate: float = 0.5*)

drop_feature(*x: torch.Tensor*, *droprate: float*)

embed(*data*)

forward(*graph: cogdl.data.data.Graph*, *x: torch.Tensor = None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

prop(*graph: cogdl.data.data.Graph*, *x: torch.Tensor*, *drop_feature_rate: float = 0.0*, *drop_edge_rate: float = 0.0*)

class cogdl.models.nn.pprgo.**PPRGo**(*in_feats*, *hidden_size*, *out_feats*, *num_layers*, *alpha*, *dropout*, *activation='relu'*, *nprop=2*, *norm='sym'*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*x*, *targets*, *ppr_scores*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*graph*, *batch_size=10000*)

class cogdl.models.nn.gin.**GIN**(*num_layers*, *in_feats*, *out_feats*, *hidden_dim*, *num_mlp_layers*, *eps=0*, *pooling='sum'*, *train_eps=False*, *dropout=0.5*)

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.pyg_dgcnn.**DGCNN**(*in_feats*, *hidden_dim*, *out_feats*, *k=20*, *dropout=0.5*)
Bases: *cogdl.models.base_model.BaseModel*

EdgeConv and DynamicGraph in paper *"Dynamic Graph CNN for Learning on Point Clouds" <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 neareast neighbors.

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

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

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.grand.**Grand**(*nfeat*, *nhid*, *nclass*, *input_droprate*, *hidden_droprate*, *use_bn*, *dropnode_rate*, *order*, *alpha*)
Bases: *cogdl.models.base_model.BaseModel*

Implementation of GRAND in paper *"Graph Random Neural Networks for Semi-Supervised Learning on Graphs"* <https://arxiv.org/abs/2005.11079>

nfeat [int] Size of each input features.

nhid [int] Size of hidden features.

nclass [int] Number of output classes.

input_droprate [float] Dropout rate of input features.

hidden_droprate [float] Dropout rate of hidden features.

use_bn [bool] Using batch normalization.

dropnode_rate [float] Rate of dropping elements of input features

tem [float] Temperature to sharpen predictions.

lam [float] Proportion of consistency loss of unlabelled data

order [int] Order of adjacency matrix

sample [int] Number of augmentations for consistency loss

alpha : float

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

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

drop_node(*x*)

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

normalize_x(*x*)

predict(*data*)

rand_prop(*graph*, *x*)

class cogdl.models.nn.pyg_gtn.**GTN**(*num_edge*, *num_channels*, *w_in*, *w_out*, *num_class*, *num_nodes*, *num_layers*)
Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

norm(*edge_index*, *num_nodes*, *edge_weight*, *improved=False*, *dtype=None*)

normalization(*H*)

class cogdl.models.nn.rgcn.**LinkPredictRGCN**(*num_entities*, *num_rels*, *hidden_size*, *num_layers*, *regularizer='basis'*, *num_bases=None*, *self_loop=True*, *sampling_rate=0.01*, *penalty=0*, *dropout=0.0*, *self_dropout=0.0*)

Bases: *cogdl.utils.link_prediction_utils.GNNLinkPredict*, *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

loss(*graph*, *scoring*)

predict(*graph*)

class cogdl.models.nn.deepergcn.**DeeperGCN**(*in_feat*, *hidden_size*, *out_feat*, *num_layers*, *activation='relu'*, *dropout=0.0*, *aggr='max'*, *beta=1.0*, *p=1.0*, *learn_beta=False*, *learn_p=False*, *learn_msg_scale=True*, *use_msg_norm=False*, *edge_attr_size=None*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*graph*)

class cogdl.models.nn.drgat.**DrGAT**(*num_features*, *num_classes*, *hidden_size*, *num_heads*, *dropout*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.models.nn.infograph.**InfoGraph**(*in_feats*, *hidden_dim*, *out_feats*, *num_layers=3*, *sup=False*)

Bases: *cogdl.models.base_model.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(*args*)
Build a new model instance.

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

classmethod split_dataset(*dataset*, *args*)

sup_forward(*batch*, *x*)

unsup_forward(*batch*, *x*)

class cogdl.models.nn.dropedge_gcn.**DropEdge_GCN**(*nfeat*, *nhid*, *nclass*, *nhidlayer*, *dropout*, *baseblock*, *inputlayer*, *outputlayer*, *nbaselayer*, *activation*, *withbn*, *withloop*, *aggrmethod*)

Bases: *cogdl.models.base_model.BaseModel*

DropEdge: Towards Deep Graph Convolutional Networks on Node Classification Applying DropEdge to GCN @ https://arxiv.org/pdf/1907.10903.pdf

The model for the single kind of deepgcn blocks. The model architecture likes: inputlayer(nfeat)–block(nbaselayer, nhid)–. . . –outputlayer(nclass)–softmax(nclass)

|—— nhidlayer —-|

The total layer is nhidlayer*nbaselayer + 2. All options are configurable.

Args: Initial function. :param nfeat: the input feature dimension. :param nhid: the hidden feature dimension. :param nclass: the output feature dimension. :param nhidlayer: the number of hidden blocks. :param dropout: the dropout ratio. :param baseblock: the baseblock type, can be "mutigcn", "resgcn", "densegcn" and "inceptiongcn". :param inputlayer: the input layer type, can be "gcn", "dense", "none". :param outputlayer: the input layer type, can be "gcn", "dense". :param nbaselayer: the number of layers in one hidden block. :param activation: the activation function, default is ReLu. :param withbn: using batch normalization in graph convolution. :param withloop: using self feature modeling in graph convolution. :param aggrmethod: the aggregation function for baseblock, can be "concat" and "add". For "resgcn", the default

is "add", for others the default is "concat".

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

reset_parameters()

class cogdl.models.nn.disengcn.**DisenGCN**(*in_feats*, *hidden_size*, *num_classes*, *K*, *iterations*, *tau*, *dropout*, *activation*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

reset_parameters()

class cogdl.models.nn.mlp.**MLP**(*in_feats*, *out_feats*, *hidden_size*, *num_layers*, *dropout=0.0*, *activation='relu'*, *norm=None*, *act_first=False*, *bias=True*)

Bases: cogdl.models.base_model.BaseModel

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

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

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.sgc.**sgc**(*in_feats*, *out_feats*)

Bases: cogdl.models.base_model.BaseModel

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.sortpool.**SortPool**(*in_feats*, *hidden_dim*, *num_classes*, *num_layers*, *out_channel*, *kernel_size*, *k=30*, *dropout=0.5*)

Bases: cogdl.models.base_model.BaseModel

Implementation 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>__.*

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(*args*)
Build a new model instance.

forward(*batch*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.pyg_srgcn.**SRGCN**(*in_feats*, *hidden_size*, *out_feats*, *attention*, *activation*, *nhop*, *normalization*, *dropout*, *node_dropout*, *alpha*, *nhead*, *subheads*)

Bases: *cogdl.models.base_model.BaseModel*

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*data*)

class cogdl.models.nn.unsup_graphsage.**SAGE**(*num_features*, *hidden_size*, *num_layers*, *sample_size*, *dropout*)

Bases: *cogdl.models.base_model.BaseModel*

Implementation of unsupervised GraphSAGE in paper *"Inductive Representation Learning on Large Graphs"* <https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf>

num_features [int] Size of each input sample

hidden_size : int num_layers : int

The number of GNN layers.

samples_size [list] The number sampled neighbors of different orders

dropout : float walk_length : int

The length of random walk

negative_samples : int

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

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

embed(*data*)

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

sampling(*edge_index*, *num_sample*)

class cogdl.models.nn.daegc.**DAEGC**(*num_features*, *hidden_size*, *embedding_size*, *num_heads*, *dropout*, *num_clusters*)
Bases: *cogdl.models.base_model.BaseModel*

The DAEGC model from the "Attributed Graph Clustering: A Deep Attentional Embedding Approach" paper

Args: num_clusters (int) : Number of clusters. T (int) : Number of iterations to recalculate P and Q gamma (float) : Hyperparameter that controls two parts of the loss.

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

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

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_2hop(*edge_index*)
add 2-hop neighbors as new edges

get_cluster_center()

get_features(*data*)

recon_loss(*z*, *adj*)

set_cluster_center(*center*)

class cogdl.models.nn.agc.**AGC**(*num_clusters*, *max_iter*, *cpu*)
Bases: *cogdl.models.base_model.BaseModel*

The AGC model from the "Attributed Graph Clustering via Adaptive Graph Convolution" paper

Args: num_clusters (int) : Number of clusters. max_iter (int) : Max iteration to increase k

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

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

compute_intra(*x*, *clusters*)

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.9.4 Model Module

cogdl.models.**build_model**(*args*)

cogdl.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

cogdl.models.**try_adding_model_args**(*model*, *parser*)

2.10 data wrappers

2.10.1 Node Classification

class cogdl.wrappers.data_wrapper.node_classification.**ClusterWrapper**(*dataset*, *method='metis'*, *batch_size=20*, *n_cluster=100*)
Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

get_train_dataset()
Return the *wrapped* dataset for specific usage. For example, return *ClusteredDataset* in cluster_dw for DDP training.

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

class cogdl.wrappers.data_wrapper.node_classification.**GraphSAGEDataWrapper**(*dataset*, *batch_size: int*, *sample_size: list*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

get_train_dataset()
Return the *wrapped* dataset for specific usage. For example, return *ClusteredDataset* in cluster_dw for DDP training.

test_wrapper()

train_transform(*batch*)

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_transform(*batch*)

val_wrapper()

class cogdl.wrappers.data_wrapper.node_classification.**M3SDataWrapper**(*dataset*, *label_rate*, *approximate*, *alpha*)

Bases: cogdl.wrappers.data_wrapper.node_classification.node_classification_dw.FullBatchNodeClfDataWrapper

static add_args(*parser*)

get_dataset()

post_stage(*stage*, *model_w_out*)
Processing after each run

pre_stage(*stage*, *model_w_out*)
Processing before each run

pre_transform()
Data Preprocessing before all runs

class cogdl.wrappers.data_wrapper.node_classification.**NetworkEmbeddingDataWrapper**(*dataset*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

class cogdl.wrappers.data_wrapper.node_classification.**FullBatchNodeClfDataWrapper**(*dataset*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

pre_transform()

Data Preprocessing before all runs

test_wrapper()

train_wrapper() → cogdl.data.data.Graph

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

class cogdl.wrappers.data_wrapper.node_classification.**PPRGoDataWrapper**(*dataset*, *topk*, *alpha=0.2*, *norm='sym'*, *batch_size=512*, *eps=0.0001*, *test_batch_size=-1*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

test_wrapper()

train_wrapper()

batch: tuple(x, targets, ppr_scores, y) x: shape=(b, num_features) targets: shape=(num_edges_of_batch,)

ppr_scores: shape=(num_edges_of_batch,) y: shape=(b, num_classes)

val_wrapper()

class cogdl.wrappers.data_wrapper.node_classification.**SAGNDataWrapper**(*dataset*, *batch_size*, *label_nhop*, *threshold*, *nhop*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

post_stage_wrapper()

pre_stage(*stage*, *model_w_out*)
Processing before each run

pre_stage_transform(*batch*)

pre_transform()
Data Preprocessing before all runs

test_transform(*batch*)

test_wrapper()

train_transform(*batch*)

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_transform(*batch*)

val_wrapper()

2.10.2 Graph Classification

class cogdl.wrappers.data_wrapper.graph_classification.**GraphClassificationDataWrapper**(*dataset*, *degree_n*, *batch_*, *train_r*, *test_ra*

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

setup_node_features()

test_wrapper()

train_wrapper()

Return:

1. DataLoader

2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

class cogdl.wrappers.data_wrapper.graph_classification.**GraphEmbeddingDataWrapper**(*dataset*, *degree_node_fe*

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

pre_transform()
Data Preprocessing before all runs

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

class cogdl.wrappers.data_wrapper.graph_classification.**InfoGraphDataWrapper**(*dataset*, *degree_node_features=*, *batch_size=32*, *train_ratio=0.5*, *test_ratio=0.3*)

Bases: cogdl.wrappers.data_wrapper.graph_classification.graph_classification_dw.GraphClassificationDataWrapper

test_wrapper()

class cogdl.wrappers.data_wrapper.graph_classification.**PATCHY_SAN_DataWrapper**(*dataset*, *num_sample*, *num_neighbor*, *stride*, **args*, ***kwargs*)

Bases: cogdl.wrappers.data_wrapper.graph_classification.graph_classification_dw.GraphClassificationDataWrapper

static add_args(*parser*)

pre_transform()
Data Preprocessing before all runs

2.10.3 Pretraining

class cogdl.wrappers.data_wrapper.pretraining.**GCCDataWrapper**(*dataset*, *batch_size*, *finetune=False*, *num_workers=4*, *rw_hops=64*, *subgraph_size=128*, *restart_prob=0.8*, *positional_embedding_size=128*, *task='node_classification'*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

2.10.4 Link Prediction

class cogdl.wrappers.data_wrapper.link_prediction.**EmbeddingLinkPredictionDataWrapper**(*dataset*, *negative_rat*

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

pre_transform()

Data Preprocessing before all runs

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

class cogdl.wrappers.data_wrapper.link_prediction.**GNNKGLinkPredictionDataWrapper**(*dataset*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Return:

1. DataLoader

2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

class cogdl.wrappers.data_wrapper.link_prediction.**GNNLinkPredictionDataWrapper**(*dataset*)
Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

pre_transform()
Data Preprocessing before all runs

test_wrapper()

static train_test_edge_split(*edge_index*, *num_nodes*, *val_ratio=0.1*, *test_ratio=0.2*)

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

2.10.5 Heterogeneous

class cogdl.wrappers.data_wrapper.heterogeneous.**HeterogeneousEmbeddingDataWrapper**(*dataset*)
Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

class cogdl.wrappers.data_wrapper.heterogeneous.**HeterogeneousGNNDataWrapper**(*dataset*)
Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

val_wrapper()

class cogdl.wrappers.data_wrapper.heterogeneous.**MultiplexEmbeddingDataWrapper**(*dataset*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Return:

1. DataLoader
2. cogdl.Graph
3. list of DataLoader or Graph

Any other data formats other than DataLoader will not be traversed

2.11 model wrappers

2.11.1 Node Classification

class cogdl.wrappers.model_wrapper.node_classification.**DGIModelWrapper**(*model*, *optimizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

static augment(*graph*)

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**GCNMixModelWrapper**(*model*, *optimizer_cfg*, *temperature*, *rampup_starts*, *rampup_ends*, *mixup_consistency*, *ema_decay*, *tau*, *k*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

GCNMixModelWrapper calls *forward_aux* in model *forward_aux* is similar to *forward* but ignores *spmm* operation.

static add_args(*parser*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

update_aux(*data*, *vector_labels*, *train_index*)

update_soft(*graph*)

val_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**GRACEModelWrapper**(*model*, *optimizer_cfg*, *tau*, *drop_feature_rates*, *drop_edge_rates*, *batch_fwd*, *proj_hidden_size*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

batched_loss(*z1: torch.Tensor*, *z2: torch.Tensor*, *batch_size: int*)

contrastive_loss(*z1: torch.Tensor*, *z2: torch.Tensor*)

prop(*graph: cogdl.data.data.Graph*, *x: torch.Tensor*, *drop_feature_rate: float = 0.0*, *drop_edge_rate: float = 0.0*)

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**GrandModelWrapper**(*model*, *optimizer_cfg*, *sample=2*, *temperature=0.5*, *lmbda=0.5*)

Bases: cogdl.wrappers.model_wrapper.node_classification.node_classification_mw.NodeClfModelWrapper

sample [int] Number of augmentations for consistency loss

temperature [float] Temperature to sharpen predictions.

lmbda [float] Proportion of consistency loss of unlabelled data

static add_args(*parser*)

consistency_loss(*logps*, *train_mask*)

train_step(*batch*)

class cogdl.wrappers.model_wrapper.node_classification.**MVGRLModelWrapper**(*model*, *optimizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**SelfAuxiliaryModelWrapper**(*model*, *optimizer_cfg*, *auxiliary_task*, *dropedge_rate*, *mask_ratio*, *sampling*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

generate_virtual_labels(*data*)

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**GraphSAGEModelWrapper**(*model*, *optimizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

val_step(*batch*)

class cogdl.wrappers.model_wrapper.node_classification.**UnsupGraphSAGEModelWrapper**(*model*, *optimizer_cfg*, *walk_length*, *negative_samples*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

setup_optimizer()

test_step(*graph*)

train_step(*batch*)

class cogdl.wrappers.model_wrapper.node_classification.**M3SModelWrapper**(*model*, *optimizer_cfg*, *n_cluster*, *num_new_labels*)

Bases: cogdl.wrappers.model_wrapper.node_classification.node_classification_mw.NodeClfModelWrapper

static add_args(*parser*)

pre_stage(*stage*, *data_w: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper*)

class cogdl.wrappers.model_wrapper.node_classification.**NetworkEmbeddingModelWrapper**(*model, num_shuf training_perce enhance=N max_eval num_wor*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser*)

test_step(*batch*)

train_step(*batch*)

class cogdl.wrappers.model_wrapper.node_classification.**NodeClfModelWrapper**(*model*, *optimizer_config*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

set_early_stopping()

Return: 1. *str*, the monitoring metric 2. tuple(*str*, *str*), that is, (the monitoring metric, *small* or *big*). The second parameter means,

the smaller, the better or *the bigger, the better*

setup_optimizer()

test_step(*batch*)

train_step(*subgraph*)

val_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**CorrectSmoothModelWrapper**(*model*, *optimizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.node_classification.node_classification_mw.NodeClfModelWrapper

static add_args(*parser*)

test_step(*batch*)

val_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**PPRGoModelWrapper**(*model*, *optimizer_config*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

val_step(*batch*)

class cogdl.wrappers.model_wrapper.node_classification.**SAGNModelWrapper**(*model*, *optimizer_config*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

val_step(*batch*)

2.11.2 Graph Classification

class cogdl.wrappers.model_wrapper.graph_classification.**GraphClassificationModelWrapper**(*mod*, *optimiz*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

val_step(*batch*)

class cogdl.wrappers.model_wrapper.graph_classification.**GraphEmbeddingModelWrapper**(*model*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

test_step(*batch*)

train_step(*batch*)

class cogdl.wrappers.model_wrapper.graph_classification.**InfoGraphModelWrapper**(*model*, *optimizer_cfg*, *sup=False*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

static mi_loss(*pos_mask*, *neg_mask*, *mi*, *pos_div*, *neg_div*)

setup_optimizer()

sup_loss(*pred*, *batch*)

test_step(*dataset*)

train_step(*batch*)

unsup_loss(*graph_feat*, *node_feat*, *batch*)

2.11.3 Pretraining

class cogdl.wrappers.model_wrapper.pretraining.**GCCModelWrapper**(*model*, *optimizer_cfg*, *nce_k*, *nce_t*, *momentum*, *output_size*, *finetune=False*, *num_classes=1*, *model_path='gcc_pretrain.pt'*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

load_checkpoint(*path*)

post_stage(*stage*, *data_w*)

pre_stage(*stage*, *data_w*)

save_checkpoint(*path*)

setup_optimizer()

train_step(*batch*)

train_step_finetune(*batch*)

train_step_pretraining(*batch*)

2.11.4 Link Prediction

class cogdl.wrappers.model_wrapper.link_prediction.**EmbeddingLinkPredictionModelWrapper**(*mode*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

test_step(*batch*)

train_step(*graph*)

class cogdl.wrappers.model_wrapper.link_prediction.**GNNKGLinkPredictionModelWrapper**(*model*, *optimizer_cfg*, *score_func*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

eval_step(*graph*, *mask1*, *mask2*)

set_early_stopping()
Return: 1. *str*, the monitoring metric 2. tuple(*str*, *str*), that is, (the monitoring metric, *small* or *big*). The second parameter means,

the smaller, the better or *the bigger, the better*

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

val_step(*subgraph*)

class cogdl.wrappers.model_wrapper.link_prediction.**GNNLinkPredictionModelWrapper**(*model*, *optimizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static get_link_labels(*num_pos*, *num_neg*, *device=None*)

set_early_stopping()
Return: 1. *str*, the monitoring metric 2. tuple(*str*, *str*), that is, (the monitoring metric, *small* or *big*). The second parameter means,

the smaller, the better or *the bigger, the better*

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

val_step(*subgraph*)

2.11.5 Heterogeneous

class cogdl.wrappers.model_wrapper.heterogeneous.**HeterogeneousEmbeddingModelWrapper**(*model*, *hidden_size=*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser: argparse.ArgumentParser*)
Add task-specific arguments to the parser.

test_step(*batch*)

train_step(*batch*)

class cogdl.wrappers.model_wrapper.heterogeneous.**HeterogeneousGNNModelWrapper**(*model*, *optimizer_config*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

val_step(*batch*)

class cogdl.wrappers.model_wrapper.heterogeneous.**MultiplexEmbeddingModelWrapper**(*model*, *hidden_size=200*, *eval_type='all'*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser: argparse.ArgumentParser*)
Add task-specific arguments to the parser.

test_step(*batch*)

train_step(*batch*)

2.11.6 Clustering

class cogdl.wrappers.model_wrapper.clustering.**AGCModelWrapper**(*model*, *optimizer_cfg*, *num_clusters*, *cluster_method='kmeans'*, *evaluation='full'*, *max_iter=5*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser*)

test_step(*batch*)

train_step(*graph*)

class cogdl.wrappers.model_wrapper.clustering.**DAEGCModelWrapper**(*model*, *optimizer_cfg*, *num_clusters*, *cluster_method='kmeans'*, *evaluation='full'*, *T=5*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

cluster_loss(*P*, *Q*)

getP(*Q*)

getQ(*z*, *cluster_center*)

post_stage(*stage*, *data_w*)

pre_stage(*stage*, *data_w*)

recon_loss(*z*, *adj*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

class cogdl.wrappers.model_wrapper.clustering.**GAEModelWrapper**(*model*, *optimizer_cfg*, *num_clusters*, *cluster_method='kmeans'*, *evaluation='full'*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

2.12 layers

class cogdl.layers.gcn_layer.**GCNLayer**(*in_features*, *out_features*, *dropout=0.0*, *activation=None*, *residual=False*, *norm=None*, *bias=True*, ***kwargs*)

Bases: torch.nn.modules.module.Module

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

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

class cogdl.layers.gat_layer.**GATLayer**(*in_feats*, *out_feats*, *nhead=1*, *alpha=0.2*, *attn_drop=0.5*, *activation=None*, *residual=False*, *norm=None*)

Bases: torch.nn.modules.module.Module

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

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

class cogdl.layers.sage_layer.**MeanAggregator**
Bases: object

class cogdl.layers.sage_layer.**SAGELayer**(*in_feats*, *out_feats*, *normalize=False*, *aggr='mean'*, *dropout=0.0*, *norm=None*, *activation=None*, *residual=False*)
Bases: torch.nn.modules.module.Module

forward(*graph*, *x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.sage_layer.**SumAggregator**
Bases: object

class cogdl.layers.gin_layer.**GINLayer**(*apply_func=None*, *eps=0*, *train_eps=True*)
Bases: torch.nn.modules.module.Module

Graph Isomorphism Network layer from paper "How Powerful are Graph Neural Networks?".

$$h_i^{(l+1)} = f_\Theta\left((1+\epsilon)h_i^l + \text{sum}\left(\left\{h_j^l, j \in \mathcal{N}(i)\right\}\right)\right)$$

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(*graph*, *x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.gcnii_layer.**GCNIILayer**(*n_channels*, *alpha=0.1*, *beta=1*, *residual=False*)
Bases: torch.nn.modules.module.Module

forward(*graph*, *x*, *init_x*)
Symmetric normalization

reset_parameters()

class cogdl.layers.deepergcn_layer.**BondEncoder**(*bond_dim_list*, *emb_size*)
Bases: torch.nn.modules.module.Module

forward(*edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.layers.deepergcn_layer.**EdgeEncoder**(*in_feats*, *out_feats*, *bias=False*)

Bases: `torch.nn.modules.module.Module`

forward(*edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.layers.deepergcn_layer.**GENConv**(*in_feats: int*, *out_feats: int*, *aggr: str = 'softmax_sg'*, *beta: float = 1.0*, *p: float = 1.0*, *learn_beta: bool = False*, *learn_p: bool = False*, *use_msg_norm: bool = False*, *learn_msg_scale: bool = True*, *norm: Optional[str] = None*, *residual: bool = False*, *activation: Optional[str] = None*, *num_mlp_layers: int = 2*, *edge_attr_size: Optional[list] = None*)

Bases: `torch.nn.modules.module.Module`

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

message_norm(*x*, *msg*)

class cogdl.layers.deepergcn_layer.**ResGNNLayer**(*conv*, *in_channels*, *activation='relu'*, *norm='batchnorm'*, *dropout=0.0*, *out_norm=None*, *out_channels=-1*, *residual=True*, *checkpoint_grad=False*)

Bases: `torch.nn.modules.module.Module`

Implementation of DeeperGCN in paper *"DeeperGCN: All You Need to Train Deeper GCNs"* <https://arxiv.org/abs/2006.07739>

conv [nn.Module] An instance of GNN Layer, recieving (graph, x) as inputs

n_channels [int] size of input features

activation : str norm: str

type of normalization, `batchnorm` as default

dropout : float checkpoint_grad : bool

forward(*graph*, *x*, *dropout=None*, **args*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.disengcn_layer.**DisenGCNLayer**(*in_feats*, *out_feats*, *K*, *iterations*, *tau=1.0*, *activation='leaky_relu'*)
Bases: `torch.nn.modules.module.Module`

Implementation of "Disentangled Graph Convolutional Networks" <http://proceedings.mlr.press/v97/ma19a.html>.

forward(*graph*, *x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

class cogdl.layers.han_layer.**AttentionLayer**(*num_features*)
Bases: `torch.nn.modules.module.Module`

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.han_layer.**HANLayer**(*num_edge*, *w_in*, *w_out*)
Bases: `torch.nn.modules.module.Module`

forward(*graph*, *x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.mlp_layer.**MLP**(*in_feats*, *out_feats*, *hidden_size*, *num_layers*, *dropout=0.0*, *activation='relu'*, *norm=None*, *act_first=False*, *bias=True*)

Bases: torch.nn.modules.module.Module

Multilayer perception with 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(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

class cogdl.layers.pprgo_layer.**LinearLayer**(*in_features*, *out_features*, *bias=True*)

Bases: torch.nn.modules.module.Module

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

class cogdl.layers.pprgo_layer.**PPRGoLayer**(*in_feats*, *hidden_size*, *out_feats*, *num_layers*, *dropout*, *activation='relu'*)

Bases: torch.nn.modules.module.Module

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.layers.rgcn_layer.**RGCNLayer**(*in_feats*, *out_feats*, *num_edge_types*, *regularizer='basis'*, *num_bases=None*, *self_loop=True*, *dropout=0.0*, *self_dropout=0.0*, *layer_norm=True*, *bias=True*)

Bases: torch.nn.modules.module.Module

Implementation of Relational-GCN in paper *"Modeling Relational Data with Graph Convolutional Networks"* <https://arxiv.org/abs/1703.06103>

in_feats [int] Size of each input embedding.

out_feats [int] Size of each output embedding.

num_edge_type [int] The number of edge type in knowledge graph.

regularizer [str, optional] Regularizer used to avoid overfitting, `basis` or `bdd`, default : `basis`.

num_bases [int, optional] The number of basis, only used when *regularizer* is *basis*, default : `None`.

self_loop [bool, optional] Add self loop embedding if True, default : `True`.

dropout : float self_dropout : float, optional

Dropout rate of self loop embedding, default : `0.0`

layer_norm [bool, optional] Use layer normalization if True, default : `True`

bias : bool

basis_forward(*graph*, *x*)

bdd_forward(*graph*, *x*)

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

reset_parameters()

Modified from https://github.com/GraphSAINT/GraphSAINT

class cogdl.layers.saint_layer.**SAINTLayer**(*dim_in*, *dim_out*, *dropout=0.0*, *act='relu'*, *order=1*, *aggr='mean'*, *bias='norm-nn'*, ***kwargs*)

Bases: torch.nn.modules.module.Module

forward(*graph*, *x*)

Inputs: graph normalized adj matrix of the subgraph x 2D matrix of input node features

Outputs: feat_out 2D matrix of output node features

class cogdl.layers.sgc_layer.**SGCLayer**(*in_features*, *out_features*, *order=3*)

Bases: torch.nn.modules.module.Module

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.layers.mixhop_layer.**MixHopLayer**(*num_features*, *adj_pows*, *dim_per_pow*)
Bases: torch.nn.modules.module.Module

adj_pow_x(*graph*, *x*, *p*)

forward(*graph*, *x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

reset_parameters()

class cogdl.layers.se_layer.**SELayer**(*in_channels*, *se_channels*)
Bases: torch.nn.modules.module.Module

Squeeze-and-excitation networks

forward(*x*)

2.13 options

cogdl.options.**add_args**(*args: list*)

cogdl.options.**add_arguments**(*args: list*)

cogdl.options.**add_data_wrapper_args**(*parser*)

cogdl.options.**add_dataset_args**(*parser*)

cogdl.options.**add_model_args**(*parser*)

cogdl.options.**add_model_wrapper_args**(*parser*)

cogdl.options.**get_default_args**(*dataset*, *model*, ***kwargs*)

cogdl.options.**get_diff_args**(*args1*, *args2*)

cogdl.options.**get_display_data_parser**()

cogdl.options.**get_download_data_parser**()

cogdl.options.**get_parser**()

cogdl.options.**get_training_parser**()

cogdl.options.**parse_args_and_arch**(*parser*, *args*)

2.14 utils

class cogdl.utils.utils.**ArgClass**
Bases: object

cogdl.utils.utils.**alias_draw**(*J*, *q*)
Draw sample from a non-uniform discrete distribution using alias sampling.

cogdl.utils.utils.**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

cogdl.utils.utils.**batch_max_pooling**(*x*, *batch*)

cogdl.utils.utils.**batch_mean_pooling**(*x*, *batch*)

cogdl.utils.utils.**batch_sum_pooling**(*x*, *batch*)

cogdl.utils.utils.**build_args_from_dict**(*dic*)

cogdl.utils.utils.**cycle_index**(*num*, *shift*)

cogdl.utils.utils.**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. name (string): saved filename. log (bool, optional): If `False`, will not print anything to the

console. (default: `True`)

cogdl.utils.utils.**get_activation**(*act: str*, *inplace=False*)

cogdl.utils.utils.**get_memory_usage**(*print_info=False*)
Get accurate gpu memory usage by querying torch runtime

cogdl.utils.utils.**get_norm_layer**(*norm: str*, *channels: int*)

Args:

norm: str type of normalization: *layernorm*, *batchnorm*, *instancenorm*

channels: int size of features for normalization

cogdl.utils.utils.**identity_act**(*input*)

cogdl.utils.utils.**makedirs**(*path*)

cogdl.utils.utils.**print_result**(*results*, *datasets*, *model_name*)

cogdl.utils.utils.**set_random_seed**(*seed*)

cogdl.utils.utils.**split_dataset_general**(*dataset*, *args*)

cogdl.utils.utils.**tabulate_results**(*results_dict*)

cogdl.utils.utils.**untar**(*path*, *fname*, *deleteTar=True*)
Unpacks the given archive file to the same directory, then (by default) deletes the archive file.

cogdl.utils.utils.**update_args_from_dict**(*args*, *dic*)

class cogdl.utils.evaluator.**Accuracy**(*mini_batch=False*)
Bases: object

clear()

evaluate()

class cogdl.utils.evaluator.**BCEWithLogitsLoss**
Bases: torch.nn.modules.module.Module

class cogdl.utils.evaluator.**BaseEvaluator**(*eval_func*)
Bases: object

clear()

evaluate()

class cogdl.utils.evaluator.**CrossEntropyLoss**
Bases: torch.nn.modules.module.Module

class cogdl.utils.evaluator.**MultiClassMicroF1**(*mini_batch=False*)
Bases: *cogdl.utils.evaluator.Accuracy*

class cogdl.utils.evaluator.**MultiLabelMicroF1**(*mini_batch=False*)
Bases: *cogdl.utils.evaluator.Accuracy*

cogdl.utils.evaluator.**accuracy**(*y_pred*, *y_true*)

cogdl.utils.evaluator.**bce_with_logits_loss**(*y_pred*, *y_true*, *reduction='mean'*)

cogdl.utils.evaluator.**cross_entropy_loss**(*y_pred*, *y_true*)

cogdl.utils.evaluator.**multiclass_f1**(*y_pred*, *y_true*)

cogdl.utils.evaluator.**multilabel_f1**(*y_pred*, *y_true*, *sigmoid=False*)

cogdl.utils.evaluator.**setup_evaluator**(*metric: Union[str, Callable]*)

class cogdl.utils.sampling.**RandomWalker**(*adj=None*, *num_nodes=None*)
Bases: object

build_up(*adj*, *num_nodes*)

walk(*start*, *walk_length*, *restart_p=0.0*)

cogdl.utils.sampling.**random_walk**

Parameters: start : np.array(dtype=np.int32) length : int indptr : np.array(dtype=np.int32) indices : np.array(dtype=np.int32) p : float

Return: list(np.array(dtype=np.int32))

cogdl.utils.graph_utils.**add_remaining_self_loops**(*edge_index*, *edge_weight=None*, *fill_value=1*, *num_nodes=None*)

cogdl.utils.graph_utils.**add_self_loops**(*edge_index*, *edge_weight=None*, *fill_value=1*, *num_nodes=None*)

cogdl.utils.graph_utils.**coalesce**(*row*, *col*, *value=None*)

cogdl.utils.graph_utils.**coo2csc**(*row*, *col*, *data*, *num_nodes=None*, *sorted=False*)

cogdl.utils.graph_utils.**coo2csr**(*row*, *col*, *data*, *num_nodes=None*, *ordered=False*)

cogdl.utils.graph_utils.**coo2csr_index**(*row*, *col*, *num_nodes=None*)

cogdl.utils.graph_utils.**csr2coo**(*indptr*, *indices*, *data*)

cogdl.utils.graph_utils.**csr2csc**(*indptr*, *indices*, *data=None*)

cogdl.utils.graph_utils.**get_degrees**(*row*, *col*, *num_nodes=None*)

cogdl.utils.graph_utils.**negative_edge_sampling**(*edge_index: Union[Tuple, torch.Tensor], num_nodes: Optional[int] = None, num_neg_samples: Optional[int] = None, undirected: bool = False*)

cogdl.utils.graph_utils.**remove_self_loops**(*indices*, *values=None*)

cogdl.utils.graph_utils.**row_normalization**(*num_nodes*, *row*, *col*, *val=None*)

cogdl.utils.graph_utils.**sorted_coo2csr**(*row*, *col*, *data*, *num_nodes=None*, *return_index=False*)

cogdl.utils.graph_utils.**symmetric_normalization**(*num_nodes*, *row*, *col*, *val=None*)

cogdl.utils.graph_utils.**to_undirected**(*edge_index*, *num_nodes=None*)

Converts the graph given by `edge_index` to an undirected graph, so that $(j, i) \in \mathcal{E}$ for every edge $(i, j) \in \mathcal{E}$.

Args: edge_index (LongTensor): The edge indices. num_nodes (int, optional): The number of nodes, *i.e.*

`max_val + 1` of `edge_index`. (default: `None`)

Return type `LongTensor`

class cogdl.utils.link_prediction_utils.**ConvELayer**(*dim*, *num_filter=20*, *kernel_size=7*, *k_w=10*, *dropout=0.3*)

Bases: `torch.nn.modules.module.Module`

concat(*ent*, *rel*)

forward(*sub_emb*, *obj_emb*, *rel_emb*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*sub_emb*, *obj_emb*, *rel_emb*)

class cogdl.utils.link_prediction_utils.**DistMultLayer**

Bases: `torch.nn.modules.module.Module`

forward(*sub_emb*, *obj_emb*, *rel_emb*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

predict(*sub_emb*, *obj_emb*, *rel_emb*)

class cogdl.utils.link_prediction_utils.**GNNLinkPredict**

Bases: `torch.nn.modules.module.Module`

forward(*graph*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_edge_set(*edge_index*, *edge_types*)

cogdl.utils.link_prediction_utils.**cal_mrr**(*embedding*, *rel_embedding*, *edge_index*, *edge_type*, *scoring*, *protocol='raw'*, *batch_size=1000*, *hits=[]*)

cogdl.utils.link_prediction_utils.**get_filtered_rank**(*heads*, *tails*, *rels*, *embedding*, *rel_embedding*, *batch_size*, *seen_data*)

cogdl.utils.link_prediction_utils.**get_rank**(*scores*, *target*)

cogdl.utils.link_prediction_utils.**get_raw_rank**(*heads*, *tails*, *rels*, *embedding*, *rel_embedding*, *batch_size*, *scoring*)

cogdl.utils.link_prediction_utils.**sampling_edge_uniform**(*edge_index*, *edge_types*, *edge_set*, *sampling_rate*, *num_rels*, *label_smoothing=0.0*, *num_entities=1*)

Args: edge_index: edge index of graph edge_types: edge_set: set of all edges of the graph, (h, t, r) sampling_rate: num_rels: label_smoothing(Optional): num_entities (Optional):

Returns: sampled_edges: sampled existing edges rels: types of smapled existing edges sampled_edges_all: existing edges with corrupted edges sampled_types_all: types of existing and corrupted edges labels: 0/1

cogdl.utils.ppr_utils.**build_topk_ppr_matrix_from_data**(*edge_index*, **args*, ***kwargs*)

cogdl.utils.ppr_utils.**calc_ppr_topk_parallel**

cogdl.utils.ppr_utils.**construct_sparse**(*neighbors*, *weights*, *shape*)

cogdl.utils.ppr_utils.**ppr_topk**(*adj_matrix*, *alpha*, *epsilon*, *nodes*, *topk*)
Calculate the PPR matrix approximately using Anderson.

cogdl.utils.ppr_utils.**topk_ppr_matrix**(*adj_matrix*, *alpha*, *eps*, *idx*, *topk*, *normalization='row'*)
Create a sparse matrix where each node has up to the topk PPR neighbors and their weights.

class cogdl.utils.prone_utils.**Gaussian**(*mu=0.5*, *theta=1*, *rescale=False*, *k=3*)
Bases: `object`

prop(*mx*, *emb*)

class cogdl.utils.prone_utils.**HeatKernel**(*t=0.5*, *theta0=0.6*, *theta1=0.4*)
Bases: `object`

prop(*mx*, *emb*)

prop_adjacency(*mx*)

class cogdl.utils.prone_utils.**HeatKernelApproximation**(*t=0.2*, *k=5*)
Bases: `object`

chebyshev(*mx*, *emb*)

prop(*mx*, *emb*)

taylor(*mx*, *emb*)

class cogdl.utils.prone_utils.**NodeAdaptiveEncoder**
Bases: object

- shrink negative values in signal/feature matrix
- no learning

static prop(*signal*)

class cogdl.utils.prone_utils.**PPR**(*alpha=0.5*, *k=10*)
Bases: object

applying sparsification to accelerate computation

prop(*mx*, *emb*)

class cogdl.utils.prone_utils.**ProNE**
Bases: object

class cogdl.utils.prone_utils.**SignalRescaling**
Bases: object

- **rescale signal of each node according to the degree of the node:**
  - sigmoid(degree)
  - sigmoid(1/degree)

prop(*mx*, *emb*)

cogdl.utils.prone_utils.**get_embedding_dense**(*matrix*, *dimension*)

cogdl.utils.prone_utils.**propagate**(*mx*, *emb*, *stype*, *space=None*)

class cogdl.utils.srgcn_utils.**ColumnUniform**
Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.utils.srgcn_utils.**EdgeAttention**(*in_feat*)
Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

class cogdl.utils.srgcn_utils.**HeatKernel**(*in_feat*)
Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.utils.srgcn_utils.**Identity**(*in_feat*)
Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.utils.srgcn_utils.**NodeAttention**(*in_feat*)
Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.utils.srgcn_utils.**NormIdentity**
Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cogdl.utils.srgcn_utils.**PPR**(*in_feat*)
Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.utils.srgcn_utils.**RowSoftmax**
Bases: `torch.nn.modules.module.Module`

forward(*edge_index*, *edge_attr*, *N*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.utils.srgcn_utils.**RowUniform**
Bases: `torch.nn.modules.module.Module`

forward(*edge_index*, *edge_attr*, *N*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class cogdl.utils.srgcn_utils.**SymmetryNorm**
Bases: `torch.nn.modules.module.Module`

forward(*edge_index*, *edge_attr*, *N*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

cogdl.utils.srgcn_utils.**act_attention**(*attn_type*)

cogdl.utils.srgcn_utils.**act_map**(*act*)

cogdl.utils.srgcn_utils.**act_normalization**(*norm_type*)

2.15 experiments

class cogdl.experiments.**AutoML**(*args*)
Bases: object

Args: search_space: function to obtain hyper-parameters to search

run()

cogdl.experiments.**auto_experiment**(*args*)

cogdl.experiments.**examine_link_prediction**(*args*, *dataset*)

cogdl.experiments.**experiment**(*dataset*, *model*, ***kwargs*)

cogdl.experiments.**gen_variants**(***items*)

cogdl.experiments.**output_results**(*results_dict*, *tablefmt='github'*)

cogdl.experiments.**raw_experiment**(*args*)

cogdl.experiments.**set_best_config**(*args*)

cogdl.experiments.**train**(*args*)

cogdl.experiments.**variant_args_generator**(*args*, *variants*)
Form variants as group with size of num_workers

2.16 pipelines

class cogdl.pipelines.**DatasetPipeline**(*app: str*, ***kwargs*)
Bases: *cogdl.pipelines.Pipeline*

class cogdl.pipelines.**DatasetStatsPipeline**(*app: str*, ***kwargs*)
Bases: *cogdl.pipelines.DatasetPipeline*

class cogdl.pipelines.**DatasetVisualPipeline**(*app: str*, ***kwargs*)
Bases: *cogdl.pipelines.DatasetPipeline*

class cogdl.pipelines.**GenerateEmbeddingPipeline**(*app: str*, *model: str*, ***kwargs*)
Bases: *cogdl.pipelines.Pipeline*

class cogdl.pipelines.**OAGBertInferencePipepline**(*app: str*, *model: str*, ***kwargs*)
Bases: *cogdl.pipelines.Pipeline*

class cogdl.pipelines.**Pipeline**(*app: str*, ***kwargs*)
Bases: object

class cogdl.pipelines.**RecommendationPipepline**(*app: str*, *model: str*, ***kwargs*)
Bases: *cogdl.pipelines.Pipeline*

cogdl.pipelines.**check_app**(*app: str*)

cogdl.pipelines.**pipeline**(*app: str*, ***kwargs*) → cogdl.pipelines.Pipeline

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y