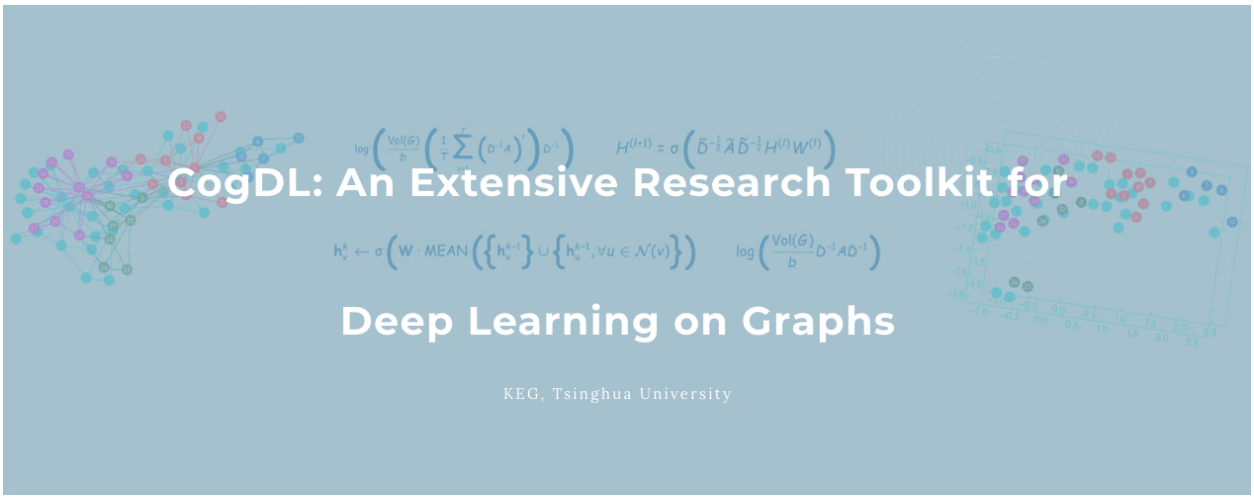
CogDL Documentation

Release 0.4.1

KEG

Aug 13, 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 **v0.4.1 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 >= 3.6
- 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), and Deep Graph Library (DGL), you can follow the instructions to install PyTorch Geometric (https://github.com/rusty1s/pytorch_geometric/#installation) and Deep Graph Library (https://docs.dgl.ai/install/index.html).

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(task="node_classification", dataset="cora", model="gcn")

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

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

# automl usage
def func_search(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(task="node_classification", dataset="cora", model="gcn", seed=[1, 2], func_
↪search=func_search)
```

2.2.2 Command-Line Usage

You can also use `python scripts/train.py --task example_task --dataset example_dataset --model example_model` to run example_model on example_data and evaluate it via example_task.

- `--task`, downstream tasks to evaluate representation like `node_classification`, `unsupervised_node_classification`, `graph_classification`. More tasks can be found in the cogdl/tasks.
- `--dataset`, dataset name to run, can be a list of datasets with space like `cora citeseer ppi`. Supported datasets include 'cora', 'citeseer', 'pumbed', 'ppi', 'wikipedia', 'blogcatalog', 'flickr'. More datasets can be found in the cogdl/datasets.
- `--model`, model name to run, can be a list of models like `deepwalk line prone`. Supported models include 'gcn', 'gat', 'graphsage', 'deepwalk', 'node2vec', 'hope', 'grarep', 'netmf', 'netsmf', 'prone'. More models can be found in the cogdl/models.

For example, if you want to run LINE, NetMF on Wikipedia with unsupervised node classification task, with 5 different seeds:

```
python scripts/train.py --task unsupervised_node_classification --dataset wikipedia --
↪model line netmf --seed 0 1 2 3 4
```

Expected output:

Variant	Micro-F1 0.1	Micro-F1 0.3	Micro-F1 0.5	Micro-F1 0.7	Micro-F1 0.9
('wikipedia', 'line')	0.4069±0.0011	0.4071±0.0010	0.4055±0.0013	0.4054±0.0020	0.4080±0.0042
('wikipedia', 'netmf')	0.4551±0.0024	0.4932±0.0022	0.5046±0.0017	0.5084±0.0057	0.5125±0.0035

If you want to run parallel experiments on your server with multiple GPUs on multiple models, GCN and GAT, on the Cora dataset with node classification task:

```
python scripts/parallel_train.py --task node_classification --dataset cora --model␣
↪gcn gat --device-id 0 1 --seed 0 1 2 3 4
```

Expected output:

Variant	Acc
('cora', 'gcn')	0.8236±0.0033
('cora', 'gat')	0.8262±0.0032

2.2.3 Fast-Spmm Usage

CogDL provides a fast sparse matrix-matrix multiplication operator called GE-SpMM to speed up training of GNN models on the GPU. You can set `fast_spmm=True` in the API usage or `--fast-spmm` in the command-line usage to enable this feature. Note that this feature is still in testing and may not work under some versions of CUDA.

2.3 Brief Tutorial

2.3.1 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.

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.

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.

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
```

2.3.2 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
```

(continues on next page)

(continued from previous page)

```
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()
```

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")
>> 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
```

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
```

(continues on next page)

(continued from previous page)

```
>> 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.])
```

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.

2.3.3 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.

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 register_model

@register_model("jknet")
class JKNet(BaseModel):
```

(continues on next page)

(continued from previous page)

```
    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)
```

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.

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)
task = build_task(args, dataset=dataset, model=model)
task.train()

# Or you may simple run the command after `register_model`
experiment(model="jknet", task="node_classification", dataset="cora")
```

2.3.4 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.

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
```

(continues on next page)

(continued from previous page)

```
        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")
```

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
```

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")
```

(continues on next page)

(continued from previous page)

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

2.4 Tasks

2.4.1 Node Classification

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

Semi-supervied Node Classification Methods

Method	Sampling	Inductive	Reproducibility
GCN			
GAT			
Chebyshev			
GraphSAGE			
GRAND			
GCNII			
DeeperGCN			
Dr-GAT			
U-net			
APPNP			
GraphMix			
DisenGCN			
SGC			
JKNet			
MixHop			
DropEdge			
SRGCN			

Tip: Reproducibility means whether the model is reproduced in our experimental setting currently.

First we define the *NodeClassification* class.

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

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

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

Then we can build dataset and model according to args. Generally the model and dataset should be placed in the same device using *.to(device)* instead of *.cuda()*. And then we set the optimizer.

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

# build model with `build_model`
model = build_model(args)
self.model = model.to(self.device)
self.patience = args.patience
self.max_epoch = args.max_epoch

# set optimizer
self.optimizer = torch.optim.Adam(
    self.model.parameters(), lr=args.lr, weight_decay=args.weight_decay
)
```

For the training process, *train* must be implemented as it will be called as the entrance of training. We provide a training loop for node classification task. For each epoch, we first call *_train_step* to optimize our model and then call *_test_step* for validation and test to compute the accuracy and loss.

```
def train(self):
    epoch_iter = tqdm(range(self.max_epoch))
    for epoch in epoch_iter:
        self._train_step()
        train_acc, _ = self._test_step(split="train")
        val_acc, val_loss = self._test_step(split="val")
        epoch_iter.set_description(
            f"Epoch: {epoch:03d}, Train: {train_acc:.4f}, Val: {val_acc:.4f}"
        )

def _train_step(self):
    """train step per epoch"""
    self.model.train()
    self.optimizer.zero_grad()
    # In node classification task, `node_classification_loss` must be defined in␣
↪model if you want to use this task directly.
    self.model.node_classification_loss(self.data).backward()
    self.optimizer.step()

def _test_step(self, split="val"):
    """test step"""
    self.model.eval()
    # `Predict` should be defined in model for inference.
    logits = self.model.predict(self.data)
    logits = F.log_softmax(logits, dim=-1)
    mask = self.data.test_mask
    loss = F.nll_loss(logits[mask], self.data.y[mask]).item()

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

In supervied node classification tasks, we use early stopping to reduce over-fitting and save training time.

```
if val_loss <= min_loss or val_acc >= max_score:
    if val_loss <= best_loss:  # and val_acc >= best_score:
        best_loss = val_loss
        best_score = val_acc
        best_model = copy.deepcopy(self.model)
    min_loss = np.min((min_loss, val_loss))
    max_score = np.max((max_score, val_acc))
    patience = 0
else:
    patience += 1
    if patience == self.patience:
        self.model = best_model
        epoch_iter.close()
        break
```

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

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

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

To run NodeClassification, we can use the following command:

```
python scripts/train.py --task node_classification --dataset cora citeseer --model␣
↪gcn gat --seed 0 1 --max-epoch 500
```

Then We get experimental results like this:

Variant	Acc
('cora', 'gcn')	0.8220±0.0010
('cora', 'gat')	0.8275±0.0015
('citeseer', 'gcn')	0.7060±0.0050
('citeseer', 'gat')	0.7060±0.0020

2.4.2 Unsupervised Node Classification

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

Unsupervied node classificatioin includes *network embedding* methods(DeepWalk, LINE, ProNE adn etc.) and *GNN self-supervied* methods(DGI, GraphSAGE and etc.). In this section, we mainly introduce the part for *network embeddings* and the other will be presented in next section *trainer*.

Unsupervised Graph Embedding Methods

Method	Weighted	shallow network	Matrix Factorization	Reproducibility	GPU support
DeepWalk					
LINE					
Node2Vec					
NetMF					
NetSMF					
HOPE					
GraRep					
SDNE					
DNGR					
ProNE					

Unsupervised Graph Neural Network Representation Learning Methods

Method	Sampling	Inductive	Reproducibility
DGI			
MVGRL			
GRACE			
GraphSAGE			

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

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

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

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

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

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

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

```
self.model = build_model(args)
self.is_weighted = self.data.edge_attr is not None


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

The spectral propagation in ProNE/ProNE++ can improve the quality of representation learned from other methods, so we can use *enhance_emb* to enhance performance. ProNE++ automatically searches for the best graph filter to help improve the embedding.

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

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

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

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

    # score each train/test group
    all_results = defaultdict(list)
    training_percents = [0.1, 0.3, 0.5, 0.7, 0.9]
    for train_percent in training_percents:
        for shuf in shuffles:
```

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

```
# ... shuffle to generate train/test set X_train/X_test, y_train/y_test

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

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

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

```
from sklearn.multiclass import OneVsRestClassifier

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

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

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

`Cogdl` supports evaluating the trained embeddings ignoring the training process. With *–load-emb-path* set to the path of your result, `Cogdl` will skip the training and directly evalute the embeddings.

The overall implementation of *UnsupervisedNodeClassification* is at (https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised_node_classification.py).

To run UnsupervisedNodeClassification, we can use following instruction:

```
python scripts/train.py --task unsupervised_node_classification --dataset ppi␣
↪wikipedia --model deepwalk prone -seed 0 1
```

Then We get experimental results like this:

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

2.4.3 Supervised Graph Classification

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

** Supervised Graph Classification Methods **

Method	Node Feature	Kernel	Reproducibility
GIN			
DiffPool			
SortPool			
PATCH_SAN			
DGCNN			
SAGPool			

Task Design

1. Set up "SupervisedGraphClassification" class, which has two specific parameters.

- *degree-feature*: Use one-hot node degree as node feature, for datasets such as lmdb-binary and lmdb-multi, which don't have node features.
- *gamma*: Multiplicative factor of learning rate decay.
- *lr*: Learning rate.

2. Build dataset convert it to a list of *Data* defined in Cogdl. Specially, we reformat the data according to the input format of specific models. *generate_data* is implemented to convert dataset.

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

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

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

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

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

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

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

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

Create a model

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

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

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

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

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

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

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

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

Run

To run GraphClassification, we can use the following command:

```
python scripts/train.py --task graph_classification --dataset proteins --model gin␣
↪diffpool sortpool dgcnn --seed 0 1
```

Then We get experimental results like this:

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

2.4.4 Unsupervised Graph Classification

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

Unsupervised Graph Classificaton Methods

Method	Node Feature	Kernel	Reproducibility
InfoGraph			
DGK			
Graph2Vec			
HGP_SL			

Task Design

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

```
@register_task("unsupervised_graph_classification")
class UnsupervisedGraphClassification(BaseTask):
    r"""Unsupervised graph classification"""
    @staticmethod
    def add_args(parser):
        """Add task-specific arguments to the parser."""
        # fmt: off
        parser.add_argument("--num-shuffle", type=int, default=10)
        parser.add_argument("--degree-feature", dest="degree_feature", action="store_
↪true")
        parser.add_argument("--lr", type=float, default=0.001)
        # fmt: on
   def __init__(self, args):
     # ...
```

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

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

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

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

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

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

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

The overall implementation of UnsupervisedGraphClassification is at (https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/unsupervised_graph_classification.py).

Create a model

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

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

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

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

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

Run

To run UnsupervisedGraphClassification, we can use the following command:

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

Then we get experimental results like this:

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

2.4.5 Link Prediction

In this tutorial, we will introduce a important link prediction. Overall speaking, the link prediction in CogDL can be divided into 3 types.

1. Network embeddings based link prediction(*HomoLinkPrediction*). All unsupervised network embedding methods supports this task for homogenous graphs without node features.
2. Knowledge graph completion(*KGLinkPrediction* and *TripleLinkPrediction*), including knowledge embedding methods(TransE, DistMult) and GNN base methods(RGCN and CompGCN).
3. GNN base homogenous graph link prediction(*GNNHomoLinkPrediction*). Theoretically, all GNN models works.

	Models
Network embeddings methods	DeepWalk, LINE, Node2Vec, ProNE NetMF, NetSMF, SDNE, Hope
Knowledge graph completion	TransE, DistMult, RotatE, RGCN, CompGCN
GNN methods	GCN and all the other GNN methods

To implement a new GNN model for link prediction, just implement *link_prediction_loss* in the model which accepting thre parameters:

- Node features.
- Edge index.
- Labels. 0/1 for each item, indicating the edge exists in the graph or is a negative sample.

The overall implementation can be found at https://github.com/THUDM/cogdl/blob/master/cogdl/tasks/link_prediction.py

2.4.6 Other Tasks

Heterogeneous Graph Embedding Methods

Method	Multi-Node	Multi-Edge	Supervised	Attribute	MetaPath
GATNE					
Metapath2Vec					
PTE					
Hin2Vec					
GTN					
HAN					

Attributed Graph Clustering

Method	Content	Spectral
kmeans		
spectral		
PRONE		
NetMF		
deepwalk		
line		
AGC		
DAEGC		

Pretrained Graph Models

- STPGNN: Strategies for pretraining graph neunral networks
- GCC: GCC: Graph Contrastive Coding for Graph Neural Network Pre-Training

2.4.7 Create new tasks

You can build a new task in the CogDL. The BaseTask class are:

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

    def __init__(self, args):
        pass

    def train(self, num_epoch):
        raise NotImplementedError
```

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

After creating your own task, you could run the task on different models and dataset. You can use 'build_model', 'build_dataset', 'build_task' method to build them with coresponding hyper-parameters.

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

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

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

    # build model, dataset and task
```

(continues on next page)

(continued from previous page)

```
    dataset = build_dataset(args)
    model = build_model(args)
    task = build_task(args)

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

2.5 Trainer

In this section, we will introduce how to implement a specific *Trainer* for a model.

In previous section, we introduce the implementation of different *tasks*. But the training paradigm varies and is incompatible with the defined training process in some cases. Therefore, *CogDL* provides *Trainer* to customize the training and inference mode. Take *NeighborSamplingTrainer* as the example, this section will show how to define a trainer.

Design

1. A self-defined trainer should inherits *BaseTrainer* and must implement function *fit* to define the training and evaluating process. Necessary parameters for training need to be added to the *add_args* in models and can be obtained here in *__init___*.

```
class NeighborSamplingTrainer(BaseTrainer):
    def __init__(self, args):
        # ... get necessary parameters from args

    def fit(self, model, dataset):
        # ... implement the training and evaluation

    @classmethod
    def build_trainer_from_args(cls, args):
        return cls(args)
```

2. All training and evaluating process, including data preprocessing and defining optimizer, should be implemented in *fit*. In other words, given the model and dataset, the rest is up to you. *fit* accepts two parameters: model and dataset, which usually are in cpu. You need to move them to cuda if you want to train on GPU.

```
def fit(self, model, dataset):
    self.data = dataset[0]

    # preprocess data
    self.train_loader = NeighborSampler(
        data=self.data,
        mask=self.data.train_mask,
        sizes=self.sample_size,
        batch_size=self.batch_size,
        num_workers=self.num_workers,
        shuffle=True,
    )
    self.test_loader = NeighborSampler(
        data=self.data, mask=None, sizes=[-1], batch_size=self.batch_size,␣
↪shuffle=False
    )
```

(continues on next page)

(continued from previous page)

```
    # move model to GPU
    self.model = model.to(self.device)

    # define optimizer
    self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.lr, weight_
↪decay=self.weight_decay)
    # training
    best_model = self.train()
    self.model = best_model
    # evaluation
    acc, loss = self._test_step()
    return dict(Acc=acc["test"], ValAcc=acc["val"])
```

3. To make the training of a model use the trainer, we should assign the trainer to the model. In Cogdl, a model must implement *get_trainer* as static method if it has a customized training process. GraphSAGE depends on *NeighborSamplingTrainer*, so the following codes should exsits in the implementation.

```
@staticmethod
def get_trainer(taskType, args):
    return NeighborSamplingTrainer
```

The details of training and evaluating are similar to the implementation in *Tasks*. The overall implementation of trainers is at https://github.com/THUDM/cogdl/tree/master/cogdl/trainers

2.6 Model

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

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

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

Then we use function decorator to declare new model for CogDL

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

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

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

@classmethod
def build_model_from_args(cls, args):
```

(continues on next page)

(continued from previous page)

```
    return cls(args.hidden_size)

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

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

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

All implemented models are at https://github.com/THUDM/cogdl/tree/master/cogdl/models.

2.7 Dataset

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

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

You should declare the name of the dataset, the name of file and the url, where our script can download resource. More implemented datasets are at https://github.com/THUDM/cogdl/tree/master/cogdl/datasets.

2.8 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*)

degrees()

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*)

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

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

remove_self_loops()

row_indptr

row_norm()

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

set_asymmetric()

set_symmetric()

subgraph(*node_idx*)

sym_norm()

test_nid

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

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

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

random_walk_with_restart(*start*, *length*, *restart_p*)

remove_self_loops()

row_indptr

row_norm()

set_symmetric(*val*)

set_weight(*weight*)

sym_norm()

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*)
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()

num_classes
The number of classes in the dataset.

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

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*)

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

static from_data_list(*data_list*)

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

len()

num_classes
The number of classes in the dataset.

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

2.9 datasets

2.9.1 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*)

2.9.2 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*

2.9.3 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*

2.9.4 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.

2.9.5 KG dataset

class cogdl.datasets.kg_data.**BidirectionalOneShotIterator**(*dataloader_head*, *dataloader_tail*)
Bases: object

static one_shot_iterator(*dataloader*)
Transform a PyTorch Dataloader into python iterator

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.**TestDataset**(*triples*, *all_true_triples*, *nentity*, *nrelation*, *mode*)
Bases: torch.utils.data.dataset.Dataset

static collate_fn(*data*)

class cogdl.datasets.kg_data.**TrainDataset**(*triples*, *nentity*, *nrelation*, *negative_sample_size*, *mode*)
Bases: torch.utils.data.dataset.Dataset

static collate_fn(*data*)

static count_frequency(*triples*, *start=4*)
Get frequency of a partial triple like (head, relation) or (relation, tail) The frequency will be used for subsampling like word2vec

static get_true_head_and_tail(*triples*)
Build a dictionary of true triples that will be used to filter these true triples for negative sampling

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*)

2.9.6 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*

2.9.7 PyG OGB dataset

class cogdl.datasets.ogb.**MAGDataset**(*data_path='data'*)
Bases: cogdl.data.dataset.Dataset

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

get_evaluator()

num_authors

num_edge_types

num_field_of_study

num_institutions

num_node_types

num_papers

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.**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.

2.9.8 PyG strategies dataset

This file is borrowed from https://github.com/snap-stanford/pretrain-gnns/

class cogdl.datasets.strategies_data.**BACEDataset**(*transform=None*, *pre_transform=None*, *pre_filter=None*, *empty=False*, *data_path='data'*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

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

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.strategies_data.**BBBPDataset**(*transform=None*, *pre_transform=None*, *pre_filter=None*, *empty=False*, *data_path='data'*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

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

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.strategies_data.**BatchAE**(*batch=None*, ***kwargs*)
Bases: cogdl.data.data.Graph

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

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

---

static from_data_list(*data_list*)
Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector `batch` is created on the fly.

num_graphs
Returns the number of graphs in the batch.

class cogdl.datasets.strategies_data.**BatchMasking**(*batch=None*, ***kwargs*)
Bases: cogdl.data.data.Graph

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*)
Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector `batch` is created on the fly.

num_graphs
Returns the number of graphs in the batch.

class cogdl.datasets.strategies_data.**BatchSubstructContext**(*batch=None*, ***kwargs*)
Bases: cogdl.data.data.Graph

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

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

---

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*)
Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector `batch` is created on the fly.

num_graphs
Returns the number of graphs in the batch.

class cogdl.datasets.strategies_data.**BioDataset**(*data_type='unsupervised'*, *empty=False*, *transform=None*, *pre_transform=None*, *pre_filter=None*, *data_path='data'*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

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

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.strategies_data.**ChemExtractSubstructureContextPair**(*k*, *l1*, *l2*)
Bases: `object`

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

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

class cogdl.datasets.strategies_data.**ExtractSubstructureContextPair**(*l1*, *center=True*)
Bases: `object`

class cogdl.datasets.strategies_data.**MoleculeDataset**(*data_type='unsupervised'*, *transform=None*, *pre_transform=None*, *pre_filter=None*, *empty=False*, *data_path='data'*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

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

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.strategies_data.**NegativeEdge**
Bases: `object`

Borrowed from https://github.com/snap-stanford/pretrain-gnns/

class cogdl.datasets.strategies_data.**TestBioDataset**(*data_type='unsupervised'*, *root='testbio'*, *transform=None*, *pre_transform=None*, *pre_filter=None*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

class cogdl.datasets.strategies_data.**TestChemDataset**(*data_type='unsupervised'*, *root='testchem'*, *transform=None*, *pre_transform=None*, *pre_filter=None*)
Bases: `cogdl.data.dataset.MultiGraphDataset`

cogdl.datasets.strategies_data.**build_batch**(*batch*, *data_list*, *num_nodes_cum*, *num_edges_cum*, *keys*)

cogdl.datasets.strategies_data.**convert**(*data*)

cogdl.datasets.strategies_data.**graph_data_obj_to_nx**(*data*)

cogdl.datasets.strategies_data.**graph_data_obj_to_nx_simple**(*data*)
Converts graph Data object required by the pytorch geometric package to network x data object. NB: Uses simplified atom and bond features, and represent as indices. NB: possible issues with recapitulating relative stereochemistry since the edges in the nx object are unordered. :param data: pytorch geometric Data object :return: network x object

cogdl.datasets.strategies_data.**nx_to_graph_data_obj**(*g*, *center_id*, *allowable_features_downstream=None*, *allowable_features_pretrain=None*, *node_id_to_go_labels=None*)

cogdl.datasets.strategies_data.**nx_to_graph_data_obj_simple**(*G*)
Converts nx graph to pytorch geometric Data object. Assume node indices are numbered from 0 to num_nodes - 1. NB: Uses simplified atom and bond features, and represent as indices. NB: possible issues with recapitulating relative stereochemistry since the edges in the nx object are unordered. :param G: nx graph obj :return: pytorch geometric Data object

cogdl.datasets.strategies_data.**reset_idxes**(*G*)
Resets node indices such that they are numbered from 0 to num_nodes - 1 :param G: :return: copy of G with relabelled node indices, mapping

2.9.9 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.**NCT109Dataset**(*data_path='data'*)
Bases: *cogdl.datasets.tu_data.TUDataset*

class cogdl.datasets.tu_data.**NCT1Dataset**(*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.**ProtainsDataset**(*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*)

cogdl.datasets.tu_data.**split**(*data*, *batch*)

2.9.10 Module contents

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

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

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

2.10 tasks

2.10.1 Base Task

class cogdl.tasks.base_task.**BaseTask**(*args*)

Bases: abc.ABC

static add_args(*parser: argparse.ArgumentParser*)

Add task-specific arguments to the parser.

get_trainer(*args*)

load_from_pretrained()

save_checkpoint()

set_evaluator(*dataset*)

set_loss_fn(*dataset*)

train()

class cogdl.tasks.base_task.**LoadFrom**

Bases: abc.ABCMeta

2.10.2 Node Classification

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

Bases: *cogdl.tasks.base_task.BaseTask*

Node classification task.

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

inference()

preprocess()

train()

2.10.3 Unsupervised Node Classification

class cogdl.tasks.unsupervised_node_classification.**TopKRanker**(*estimator*, ***, *n_jobs=None*)

Bases: sklearn.multiclass.OneVsRestClassifier

predict(*X*, *top_k_list*)
Predict multi-class targets using underlying estimators.

X [(sparse) array-like of shape (n_samples, n_features)] Data.

y [(sparse) array-like of shape (n_samples,) or (n_samples, n_classes)] Predicted multi-class targets.

class cogdl.tasks.unsupervised_node_classification.**UnsupervisedNodeClassification**(*args*, *dataset=Nor*
model=None)

Bases: *cogdl.tasks.base_task.BaseTask*

Node classification task.

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

enhance_emb(*G*, *embs*)

save_emb(*embs*)

train()

2.10.4 Heterogeneous Node Classification

class cogdl.tasks.heterogeneous_node_classification.**HeterogeneousNodeClassification**(*args*, *dataset=*
model=N

Bases: *cogdl.tasks.base_task.BaseTask*

Heterogeneous Node classification task.

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

train()

2.10.5 Multiplex Node Classification

class cogdl.tasks.multiplex_node_classification.**MultiplexNodeClassification**(*args*, *dataset=None*, *model=None*)

Bases: *cogdl.tasks.base_task.BaseTask*

Node classification task.

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

train()

2.10.6 Link Prediction

class cogdl.tasks.link_prediction.**GNNHomoLinkPrediction**(*args*, *dataset=None*, *model=None*)

Bases: torch.nn.modules.module.Module

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

train()
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

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

class cogdl.tasks.link_prediction.**HomoLinkPrediction**(*args*, *dataset=None*, *model=None*)

Bases: torch.nn.modules.module.Module

train()
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.tasks.link_prediction.**KGLinkPrediction**(*args*, *dataset=None*, *model=None*)

Bases: torch.nn.modules.module.Module

train()
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.tasks.link_prediction.**LinkPrediction**(*args*, *dataset=None*, *model=None*)

Bases: *cogdl.tasks.base_task.BaseTask*

static add_args(*parser*)

Add task-specific arguments to the parser.

load_from_pretrained()

save_checkpoint()

train()

class cogdl.tasks.link_prediction.**TripleLinkPrediction**(*args*, *dataset=None*, *model=None*)

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

Training process borrowed from *KnowledgeGraphEmbedding<https://github.com/DeepGraphLearning/KnowledgeGraphEmbeda*

test_step(*model*, *test_triples*, *all_true_triples*, *args*)

Evaluate the model on test or valid datasets

train()

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

train_step(*model*, *optimizer*, *train_iterator*, *args*)

A single train step. Apply back-propation and return the loss

cogdl.tasks.link_prediction.**divide_data**(*input_list*, *division_rate*)

cogdl.tasks.link_prediction.**evaluate**(*embs*, *true_edges*, *false_edges*)

cogdl.tasks.link_prediction.**gen_node_pairs**(*train_data*, *test_data*, *negative_ratio=5*)

cogdl.tasks.link_prediction.**get_score**(*embs*, *node1*, *node2*)

cogdl.tasks.link_prediction.**log_metrics**(*mode*, *step*, *metrics*)

Print the evaluation logs

cogdl.tasks.link_prediction.**randomly_choose_false_edges**(*nodes*, *true_edges*, *num*)

cogdl.tasks.link_prediction.**save_model**(*model*, *optimizer*, *save_variable_list*, *args*)

Save the parameters of the model and the optimizer, as well as some other variables such as step and learning_rate

cogdl.tasks.link_prediction.**select_task**(*model_name=None*, *model=None*)

cogdl.tasks.link_prediction.**set_logger**(*args*)

Write logs to checkpoint and console

2.10.7 Multiplex Link Prediction

class cogdl.tasks.multiplex_link_prediction.**MultiplexLinkPrediction**(*args*, *dataset=None*, *model=None*)

Bases: *cogdl.tasks.base_task.BaseTask*

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

train()

cogdl.tasks.multiplex_link_prediction.**evaluate**(*embs*, *true_edges*, *false_edges*)

cogdl.tasks.multiplex_link_prediction.**get_score**(*embs*, *node1*, *node2*)

2.10.8 Graph Classification

class cogdl.tasks.graph_classification.**GraphClassification**(*args*, *dataset=None*, *model=None*)

Bases: *cogdl.tasks.base_task.BaseTask*

Superiviced graph classification task.

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

train()

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

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

2.10.9 Unsupervised Graph Classification

class cogdl.tasks.unsupervised_graph_classification.**UnsupervisedGraphClassification**(*args*, *dataset=*
model=N

Bases: *cogdl.tasks.base_task.BaseTask*

Unsupervised graph classification

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

save_emb(*embs*)

train()

2.10.10 Attributed Graph Clustering

class cogdl.tasks.attributed_graph_clustering.**AttributedGraphClustering**(*args*, *dataset=None*, *_=None*)

Bases: *cogdl.tasks.base_task.BaseTask*

Attributed graph clustring task.

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

train() → Dict[str, float]

2.10.11 Similarity Search

class cogdl.tasks.similarity_search.**SimilaritySearch**(*args*, *dataset=None*, *model=None*)
Bases: *cogdl.tasks.base_task.BaseTask*

Similarity Search task.

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

train()

2.10.12 Pretrain

class cogdl.tasks.pretrain.**PretrainTask**(*args*)
Bases: *cogdl.tasks.base_task.BaseTask*

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

train()

2.10.13 Task Module

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

cogdl.tasks.**register_task**(*name*)
New task types can be added to cogdl with the *register_task()* function decorator.

For example:

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

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

cogdl.tasks.**try_import_task**(*task*)

2.11 models

2.11.1 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.

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.

---

static get_trainer(*args=None*) → Optional[Type[cogdl.trainers.base_trainer.BaseTrainer]]

graph_classification_loss(*batch*)

node_classification_loss(*data*, *mask=None*)

predict(*data*)

set_device(*device*)

set_loss_fn(*loss_fn*)

2.11.2 Supervised Model

class cogdl.models.supervised_model.**SupervisedHeterogeneousNodeClassificationModel**
Bases: *cogdl.models.base_model.BaseModel*, abc.ABC

evaluate(*data: Any*, *nodes: Any*, *targets: Any*) → Any

static get_trainer(*args: Any = None*) → Optional[Type[SupervisedHeterogeneousNodeClassificationTrainer]]

loss(*data: Any*) → Any

class cogdl.models.supervised_model.**SupervisedHomogeneousNodeClassificationModel**
Bases: *cogdl.models.base_model.BaseModel*, abc.ABC

static get_trainer(*args: Any = None*) → Optional[Type[SupervisedHomogeneousNodeClassificationTrainer]]

loss(*data: Any*) → Any

predict(*data: Any*) → Any

class cogdl.models.supervised_model.**SupervisedModel**
Bases: *cogdl.models.base_model.BaseModel*, abc.ABC

loss(*data: Any*) → Any

2.11.3 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.

model_name = 'hope'

train(*G*)
The author claim that Katz has superior performance in related tasks S_katz = (M_g)^-1 * M_l = (I - beta*A)^-1 * beta*A = (I - beta*A)^-1 * (I - (I -beta*A)) = (I - beta*A)^-1 - I

class cogdl.models.emb.spectral.**Spectral**(*dimension*)
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.

model_name = 'spectral'

train(*G*)
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.

model_name = 'hin2vec'

train(*G*, *node_type*)
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.

model_name = 'netmf'

train(*G*)

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.distmult.**DistMult**(*nentity*, *nrelation*, *hidden_dim*, *gamma*, *double_entity_embedding=False*, *double_relation_embedding=False*)

Bases: cogdl.models.emb.knowledge_base.KGEModel

The DistMult model from the ICLR 2015 paper *"EMBEDDING ENTITIES AND RELATIONS FOR LEARNING AND INFERENCE IN KNOWLEDGE BASES" <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ICLR2015_updated.pdf>* borrowed from *KnowledgeGraphEmbedding<https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding>*

model_name = 'distmult'

score(*head*, *relation*, *tail*, *mode*)

class cogdl.models.emb.transe.**TransE**(*nentity*, *nrelation*, *hidden_dim*, *gamma*, *double_entity_embedding=False*, *double_relation_embedding=False*)

Bases: cogdl.models.emb.knowledge_base.KGEModel

The TransE model from paper *"Translating Embeddings for Modeling Multi-relational Data" <http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf>* borrowed from *KnowledgeGraphEmbedding<https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding>*

model_name = 'transe'

score(*head*, *relation*, *tail*, *mode*)

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.

model_name = 'deepwalk'

train(*G: networkx.classes.graph.Graph*, *embedding_model_creator=<class 'gensim.models.word2vec.Word2Vec'>*)
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.rotate.**RotatE**(*nentity*, *nrelation*, *hidden_dim*, *gamma*, *double_entity_embedding=False*, *double_relation_embedding=False*)

Bases: `cogdl.models.emb.knowledge_base.KGEModel`

Implementation of RotatE model from the paper *"RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space" <https://openreview.net/forum?id=HkgEQnRqYQ>*. borrowed from *KnowledgeGraphEmbedding<https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding>*

model_name = 'rotate'

score(*head*, *relation*, *tail*, *mode*)

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.

model_name = 'gatne'

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.

---

model_name = 'dgk'

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.

model_name = 'grarep'

train(*G*)
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.

get_denoised_matrix(*mat*)

get_emb(*matrix*)

get_ppmi_matrix(*mat*)

model_name = 'dngr'

random_surfing(*adj_matrix*)

scale_matrix(*mat*)

train(*G*)
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.

model_name = 'prone++'

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.

---

model_name = 'graph2vec'

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.

model_name = 'metapath2vec'

train(*G*, *node_type*)
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.

model_name = 'node2vec'

train(*G*)
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.complex.**ComplEx**(*nentity*, *nrelation*, *hidden_dim*, *gamma*, *double_entity_embedding=False*, *double_relation_embedding=False*)
Bases: `cogdl.models.emb.knowledge_base.KGEModel`

the implementation of ComplEx model from the paper *"Complex Embeddings for Simple Link Prediction"<http://proceedings.mlr.press/v48/trouillon16.pdf>* borrowed from *KnowledgeGraphEmbedding<https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding>*

model_name = 'complex'

score(*head*, *relation*, *tail*, *mode*)

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.

model_name = 'pte'

train(*G*, *node_type*)
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.

model_name = 'netsmf'

train(*G*)
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.

model_name = 'line'

train(*G*)
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.

model_name = 'sdne'

train(*G*)
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.

model_name = 'prone'

train(*G*)
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

2.11.4 GNN Model

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

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.

---

static get_trainer(*args*)

loss(*data*)

model_name = 'dgi'

self_supervised_loss(*data*)

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

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.

---

static get_trainer(*args*)

loss(*data*)

model_name = 'mvgrl'

preprocess(*graph*)

self_supervised_loss(*data*)

class cogdl.models.nn.patchy_san.**PatchySAN**(*batch_size*, *num_features*, *num_classes*, *num_sample*, *stride*, *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.

---

model_name = 'patchy_san'

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.

---

model_name = 'chebyshev'

predict(*data*)

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

---

model_name = 'gcn'

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.

---

model_name = 'gdc_gcn'

node_classification_loss(*data*)

predict(*data=None*)

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

reset_data(*data*)

class cogdl.models.nn.pyg_hgpsl.**HGPSL**(*num_features*, *num_classes*, *hidden_size*, *dropout*, *pooling*, *sample_neighbor*, *sparse_attention*, *structure_learning*, *lamb*)
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(*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.

---

model_name = 'hgpsl'

classmethod split_dataset(*dataset*, *args*)

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.

static get_trainer(*args*)

inference(*x_all*, *data_loader*)

mini_forward(*graph*)

mini_loss(*data*)

model_name = 'graphsage'

node_classification_loss(**args*)

predict(*data*)

sampling(*edge_index*, *num_sample*)

set_data_device(*device*)

class cogdl.models.nn.compgcn.**LinkPredictCompGCN**(*num_entities*, *num_rels*, *hidden_size*, *num_bases=0*, *layers=1*, *sampling_rate=0.01*, *score_func='conve'*, *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*, *split='train'*)

model_name = 'compgcn'

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.

---

model_name = 'drgcn'

predict(*graph*)

class cogdl.models.nn.pyg_gpt_gnn.**GPT_GNN**
Bases: *cogdl.models.supervised_model.SupervisedHomogeneousNodeClassificationModel*, *cogdl.models.supervised_model.SupervisedHeterogeneousNodeClassificationModel*

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

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

evaluate(*data: Any*, *nodes: Any*, *targets: Any*) → Any

static get_trainer(*args*) → Optional[Type[Union[cogdl.trainers.gpt_gnn_trainer.GPT_GNNHomogeneousTrainer, cogdl.trainers.gpt_gnn_trainer.GPT_GNNHeterogeneousTrainer]]]

loss(*data: Any*) → Any

model_name = 'gpt_gnn'

predict(*data: Any*) → Any

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.

model_name = 'unet'

class cogdl.models.nn.gcnmix.**GCNMix**(*in_feat*, *hidden_size*, *num_classes*, *k*, *temperature*, *alpha*, *rampup_starts*, *rampup_ends*, *final_consistency_weight*, *ema_decay*, *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_ema(*graph*)

model_name = 'gcnmix'

node_classification_loss(*data*)

predict(*data*)

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*)

model_name = 'diffpool'

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*)

model_name = 'gcnii'

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.

model_name = 'mlp'

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*)

model_name = 'pyg_gcn'

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.

---

model_name = 'mixhop'

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.

---

model_name = 'gat'

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.

evaluate(*data*, *nodes*, *targets*)

forward(*graph*, *target_x*, *target*)
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*)

model_name = 'han'

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.

---

model_name = 'ppnp'

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.self_supervised_model.SelfSupervisedContrastiveModel

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*)
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.

---

static get_trainer(*args*)

model_name = 'grace'

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

self_supervised_loss(*graph*)

class cogdl.models.nn.dgl_jknet.**JKNet**(*in_features*, *out_features*, *n_layers*, *n_units*, *node_aggregation*, *layer_aggregation*)
Bases: *cogdl.models.supervised_model.SupervisedHomogeneousNodeClassificationModel*

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

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

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.

---

static get_trainer(*args*)

loss(*data*)

model_name = 'jknet'

predict(*data*)

set_graph(*graph*)

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

static get_trainer(*args: Any*)

model_name = 'pprgo'

node_classification_loss(*x*, *targets*, *ppr_scores*, *y*)

predict(*graph*, *batch_size*, *norm*)

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.

model_name = 'gin'

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.

---

model_name = 'dgcnn'

classmethod split_dataset(*dataset*, *args*)

class cogdl.models.nn.grand.**Grand**(*nfeat*, *nhid*, *nclass*, *input_droprate*, *hidden_droprate*, *use_bn*, *dropnode_rate*, *tem*, *lam*, *order*, *sample*, *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.

consis_loss(*logps*, *train_mask*)

dropNode(*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.

---

model_name = 'grand'

node_classification_loss(*graph*)

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.

evaluate(*data*, *nodes*, *targets*)

forward(*graph*, *target_x*, *target*)
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*)

model_name = 'gtn'

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*, *split='train'*)

model_name = 'rgcn'

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.

---

model_name = 'deepergcn'

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.

model_name = 'drgat'

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.

graph_classification_loss(*batch*)

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

model_name = 'infograph'

reset_parameters()

classmethod split_dataset(*dataset*, *args*)

sup_forward(*batch*, *x*)

sup_loss(*pred*, *batch*)

unsup_forward(*batch*, *x*)

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

unsup_sup_loss(*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.

---

model_name = 'dropedge_gcn'

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.

model_name = 'disengcn'

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.

model_name = 'mlp'

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.

model_name = 'sgc'

predict(*data*)

class cogdl.models.nn.stpgnn.**stpgnn**(*args*)

Bases: *cogdl.models.base_model.BaseModel*

Implementation of models in paper *"Strategies for Pre-training Graph Neural Networks"*. <https://arxiv.org/abs/1905.12265>

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

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

model_name = 'stpgnn'

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*

Implimentation of sortpooling in paper *"An End-to-End Deep Learning Architecture for Graph Classification" <https://www.cse.wustl.edu/~muhan/papers/AAAI_2018_DGCNN.pdf>__.*

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.

---

model_name = 'sortpool'

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.

---

model_name = 'srgcn'

predict(*data*)

class cogdl.models.nn.dgl_gcc.**GCC**(*load_path*)
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.

model_name = 'gcc'

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.nn.unsup_graphsage.**SAGE**(*num_features*, *hidden_size*, *num_layers*, *sample_size*, *dropout*, *walk_length*, *negative_samples*)
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.

---

static get_trainer(*args*)

loss(*data*)

model_name = 'unsup_graphsage'

node_classification_loss(*data*)

sampling(*edge_index*, *num_sample*)

self_supervised_loss(*data*)

class cogdl.models.nn.pyg_sagpool.**SAGPoolNetwork**(*nfeat*, *nhid*, *nclass*, *dropout*, *pooling_ratio*, *pooling_layer_type*)
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(*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.

---

model_name = 'sagpool'

classmethod split_dataset(*dataset*, *args*)

2.11.5 AGC Model

2.11.6 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_import_model**(*model*)

2.12 layers

2.12.1 Layers

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

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_features*, *out_features*, *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*)

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*)

2.12.2 GCC module

2.12.3 GPT-GNN module

2.12.4 Link Prediction module

2.12.5 PPRGo module

2.12.6 ProNE module

2.12.7 SRGCN module

2.12.8 Strategies module

2.13 options

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

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

cogdl.options.**add_task_args**(*parser*)

cogdl.options.**add_trainer_args**(*parser*)

cogdl.options.**get_default_args**(*task: str*, *dataset*, *model*, ***kwargs*)

cogdl.options.**get_display_data_parser**()

cogdl.options.**get_download_data_parser**()

cogdl.options.**get_parser**()

cogdl.options.**get_task_model_args**(*task*, *model=None*)

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*)

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*, *inplace=True*)

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*)

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*)

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))

2.15 experiments

class cogdl.experiments.**AutoML**(*task*, *dataset*, *model*, *n_trials=3*, ***kwargs*)
Bases: object

Args: func_search: function to obtain hyper-parameters to search

run()

cogdl.experiments.**auto_experiment**(*task: str*, *dataset*, *model*, ***kwargs*)

cogdl.experiments.**check_task_dataset_model_match**(*task*, *variants*)

cogdl.experiments.**experiment**(*task: str*, *dataset*, *model*, ***kwargs*)

cogdl.experiments.**gen_variants**(***items*)

cogdl.experiments.**output_results**(*results_dict*, *tablefmt='github'*)

cogdl.experiments.**raw_experiment**(*task: str*, *dataset*, *model*, ***kwargs*)

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

J

K

L

M

N

O

P

R

S

T

U

V

W

Y