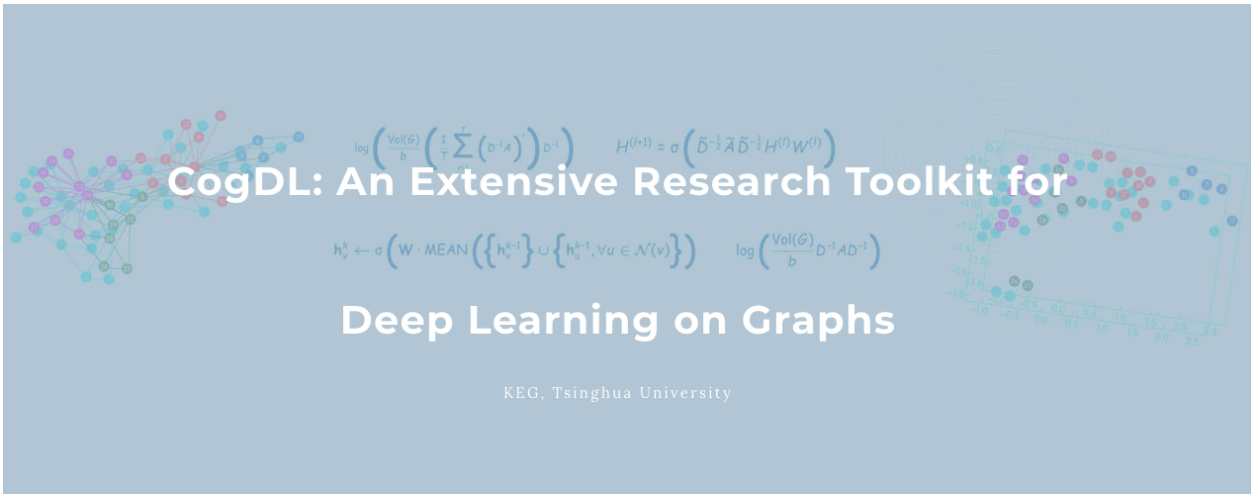
CogDL Documentation

Release 0.6

KEG

Apr 27, 2023

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:

- **Efficiency**: CogDL utilizes well-optimized operators to speed up training and save GPU memory of GNN models.
- **Ease of 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.

CHAPTER

ONE

NEWS

- [The CogDL paper](https://arxiv.org/abs/2103.00959) was accepted by [WWW 2023](https://www2023.thewebconf.org/). Find us at WWW 2023! We also release the new **v0.6 release** which adds more examples of graph self-supervised learning, including [GraphMAE](https://github.com/THUDM/cogdl/tree/master/examples/graphmae), [GraphMAE2](https://github.com/THUDM/cogdl/tree/master/examples/graphmae2), and [BGRL](https://github.com/THUDM/cogdl/tree/master/examples/bgrl).
- The new **v0.5.3 release** supports mixed-precision training by setting textit{fp16=True} and provides a basic [example](https://github.com/THUDM/cogdl/blob/master/examples/jittor/gcn.py) written by [Jittor](https://github.com/Jittor/jittor). It also updates the tutorial in the document, fixes downloading links of some datasets, and fixes potential bugs of operators.
- The new **v0.5.2 release** adds a GNN example for ogbn-products and updates geom datasets. It also fixes some potential bugs including setting devices, using cpu for inference, etc.
- The new **v0.5.1 release** adds fast operators including SpMM (cpu version) and scatter_max (cuda version). It also adds lots of datasets for node classification.
- The new **v0.5.0 release** designs and implements a unified training loop for GNN. It introduces *DataWrapper* to help prepare the training/validation/test data and *ModelWrapper* to define the training/validation/test steps.
- The new **v0.4.1 release** adds the implementation of Deep GNNs and the recommendation task. It also supports new pipelines for generating embeddings and recommendation. Welcome to join our tutorial on KDD 2021 at 10:30 am - 12:00 am, Aug. 14th (Singapore Time). More details can be found in https://kdd2021graph.github.io/.
- The new **v0.4.0 release** refactors the data storage (from `Data` to `Graph`) and provides more fast operators to speed up GNN training. It also includes many self-supervised learning methods on graphs. BTW, we are glad to announce that we will give a tutorial on KDD 2021 in August. Please see this link for more details.
- The new **v0.3.0 release** provides a fast spmm operator to speed up GNN training. We also release the first version of CogDL paper in arXiv. You can join our slack for discussion.
- The new **v0.2.0 release** includes easy-to-use `experiment` and `pipeline` APIs for all experiments and applications. The `experiment` API supports automl features of searching hyper-parameters. This release also

provides `OAGBert` API for model inference (`OAGBert` is trained on large-scale academic corpus by our lab). Some features and models are added by the open source community (thanks to all the contributors).

- The new **v0.1.2 release** includes a pre-training task, many examples, OGB datasets, some knowledge graph embedding methods, and some graph neural network models. The coverage of CogDL is increased to 80%. Some new APIs, such as `Trainer` and `Sampler`, are developed and being tested.
- The new **v0.1.1 release** includes the knowledge link prediction task, many state-of-the-art models, and `optuna` support. We also have a Chinese WeChat post about the CogDL release.

CHAPTER

TWO

CITING COGDL

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

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

2.1 Install

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

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

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

```
pip install cogdl
```

Install from source via:

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

Or clone the repository and install with the following commands:

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

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

2.2 Quick Start

2.2.1 API Usage

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

```
from cogdl import experiment

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

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

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

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

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

2.2.2 Command-Line Usage

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

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

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

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

Expected output:

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

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

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

Expected output:

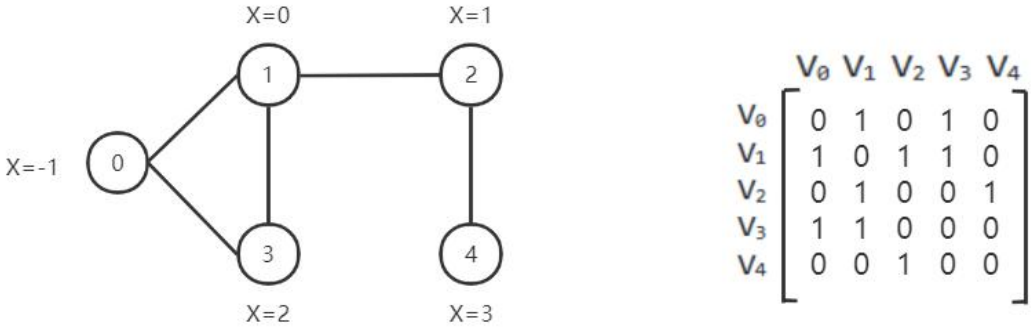
Variant	test_acc	val_acc
('cora', 'gcn')	0.8050±0.0047	0.7940±0.0063
('cora', 'gat')	0.8234±0.0042	0.8088±0.0016
('citeseer', 'gcn')	0.6938±0.0133	0.7108±0.0148
('citeseer', 'gat')	0.7098±0.0053	0.7244±0.0039

2.3 Introduction to Graphs

2.3.1 Real-world graphs

Graph-structured data have been widely utilized in many real-world scenarios. For example, each user on Facebook can be seen as a vertex and their relations like friendship or followership can be seen as edges in the graph. We might be interested in predicting the interests of users, or whether a pair of nodes in a network might have an edge connecting them.

A graph can be represented using an adjacency matrix



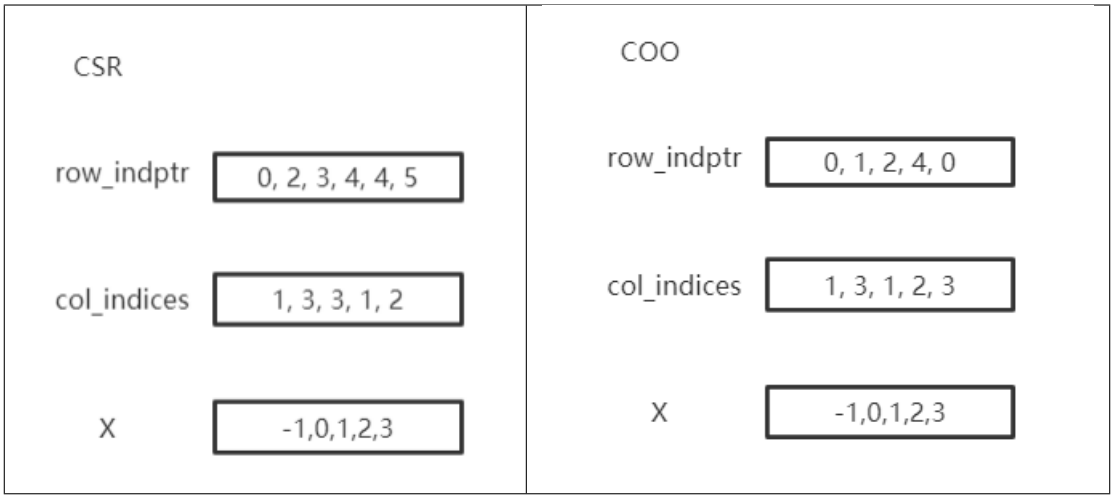
2.3.2 How to represent a graph in CogDL

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.g., *graph.mask = mask*, are also supported.

Represent this graph in cogdl:



`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()
x = torch.tensor([[-1],[0],[1],[2],[3]])
g = Graph(edge_index=edges,x=x) # equivalent to that above
print(g.row_indptr)
>>tensor([0, 2, 3, 4, 4, 5])
print(g.col_indices)
>>tensor([1, 3, 3, 1, 2])
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*:

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

- `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
>> 2708
g.num_edges
>> 10556
# 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
```

(continues on next page)

(continued from previous page)

```
model.eval()
graph.eval()
```

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

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

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

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

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

2.3.4 How to edit the graph?

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
>> 10556
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
>> 10556

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

2.3.5 Common graph datasets

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

```
from cogdl.datasets import build_dataset_from_name
dataset = build_dataset_from_name("cora")

from cogdl.datasets import build_dataset
dataset = build_dataset(args) # if 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.

CogDL now supports the following datasets for different tasks:

- Network Embedding (Unsupervised node classification): PPI, Blogcatalog, Wikipedia, Youtube, DBLP, Flickr
- Semi/Un-superviesd Node classification: Cora, Citeseer, Pubmed, Reddit, PPI, PPI-large, Yelp, Flickr, Amazon
- Heterogeneous node classification: DBLP, ACM, IMDB

- Link prediction: PPI, Wikipedia, Blogcatalog
- Multiplex link prediction: Amazon, YouTube, Twitter
- graph classification: MUTAG, IMDB-B, IMDB-M, PROTEINS, COLLAB, NCI, NCI109, Reddit-BINARY

Network Embedding(Unsupervised Node classification)

Dataset	Nodes	Edges	Classes	Degree	Name in Cogdl
PPI	3,890	76,584	50(m)	—	ppi-ne
BlogCatalog	10,312	333,983	40(m)	32	blogcatalog
Wikipedia	4.777	184,812	39(m)	39	wikipedia
Flickr	80,513	5,899,882	195(m)	73	flickr-ne
DBLP	51,264	2,990,443	60(m)	2	dblp-ne
Youtube	1,138,499	2,990,443	47(m)	3	youtube-ne

Node classification

Dataset	Nodes	Edges	Features	Classes	Train/Val/Test	Degree	Name in cogdl
Cora	2,708	5,429	1,433	7(s)	140 / 500 / 1000	2	cora
Citeseer	3,327	4,732	3,703	6(s)	120 / 500 / 1000	1	citeseer
PubMed	19,717	44,338	500	3(s)	60 / 500 / 1999	2	pubmed
Chameleon	2,277	36,101	2,325	5	0.48 / 0.32 / 0.20	16	chameleon
Cornell	183	298	1,703	5	0.48 / 0.32 / 0.20	1.6	cornell
Film	7,600	30,019	932	5	0.48 / 0.32 / 0.20	4	film
Squirrel	5201	217,073	2,089	5	0.48 / 0.32 / 0.20	41.7	squirrel
Texas	182	325	1,703	5	0.48 / 0.32 / 0.20	1.8	texas
Wisconsin	251	515	1,703	5	0.48 / 0.32 / 0.20	2	Wisconsin
PPI	14,755	225,270	50	121(m)	0.66 / 0.12 / 0.22	15	ppi
PPI-large	56,944	818,736	50	121(m)	0.79 / 0.11 / 0.10	14	ppi-large
Reddit	232,965	11,606,919	602	41(s)	0.66 / 0.10 / 0.24	50	reddit
Flickr	89,250	899,756	500	7(s)	0.50 / 0.25 / 0.25	10	flickr
Yelp	716,847	6,977,410	300	100(m)	0.75 / 0.10 / 0.15	10	yelp
Amazon-SAINT	1,598,960	132,169,734	200	107(m)	0.85 / 0.05 / 0.10	83	amazon-s

Heterogenous Graph

Dataset	Nodes	Edges	Features	Classes	Train/Val/Test	Degree	Edge Type	Name in Cogdl
DBLP	18,405	67,946	334	4	800 / 400 / 2857	4	4	gtn-dblp(han-acm)
ACM	8,994	25,922	1,902	3	600 / 300 / 2125	3	4	gtn-acm(han-acm)
IMDB	12,772	37,288	1,256	3	300 / 300 / 2339	3	4	gtn-imdb(han-imdb)
Amazon-GATNE	10,166	148,863	—	—	—	15	2	amazon
Youtube-GATNE	2,000	1,310,617	—	—	—	655	5	youtube
Twitter	10,000	331,899	—	—	—	33	4	twitter

Knowledge Graph Link Prediction

Dataset	Nodes	Edges	Train/Val/Test	Relations Types	Degree	Name in Cogdl
FB13	75,043	345,872	316,232 / 5,908 / 23,733	12	5	fb13
FB15k	14,951	592,213	483,142 / 50,000 / 59,071	1345	40	fb15k
FB15k-237	14,541	310,116	272,115 / 17,535 / 20,466	237	21	fb15k237
WN18	40,943	151,442	141,442 / 5,000 / 5,000	18	4	wn18
WN18RR	86,835	93,003	86,835 / 3,034 / 3,134	11	1	wn18rr

Graph Classification

TUdataset from https://www.chrsmrrs.com/graphkerneldatasets

Dataset	Graphs	Classes	Avg. Size	Name in Cogdl
MUTAG	188	2	17.9	mutag
IMDB-B	1,000	2	19.8	imdb-b
IMDB-M	1,500	3	13	imdb-m
PROTEINS	1,113	2	39.1	proteins
COLLAB	5,000	5	508.5	collab
NCI1	4,110	2	29.8	nci1
NCI109	4,127	2	39.7	nci109
PTC-MR	344	2	14.3	ptc-mr
REDDIT-BINARY	2,000	2	429.7	reddit-b
REDDIT-MULTI-5k	4,999	5	508.5	reddit-multi-5k
REDDIT-MULTI-12k	11,929	11	391.5	reddit-multi-12k
BBBP	2,039	2	24	bbbp
BACE	1,513	2	34.1	bace

2.4 Models of Cogdl

2.4.1 Introduction to graph representation learning

Inspired by recent trends of representation learning on computer vision and natural language processing, graph representation learning is proposed as an efficient technique to address this issue. Graph representation aims at either learning low-dimensional continuous vectors for vertices/graphs while preserving intrinsic graph properties, or using graph encoders to an end-to-end training.

Recently, graph neural networks (GNNs) have been proposed and have achieved impressive performance in semi-supervised representation learning. Graph Convolution Networks (GCNs) proposes a convolutional architecture via a localized first-order approximation of spectral graph convolutions. GraphSAGE is a general inductive framework that leverages node features to generate node embeddings for previously unseen samples. Graph Attention Networks (GATs) utilizes the multi-head self-attention mechanism and enables (implicitly) specifying different weights to different nodes in a neighborhood.

2.4.2 CogDL now supports the following tasks

- unsupervised node classification
- semi-supervised node classification
- heterogeneous node classification
- link prediction
- multiplex link prediction

- unsupervised graph classification
- supervised graph classification
- graph pre-training
- attributed graph clustering

CogDL provides abundant of common benchmark datasets and GNN models. You can simply start a running using models and datasets in CogDL.

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

Unsupervised Multi-label Node Classification

Model	Name in Cogdl
NetMF (Qiu et al, WSDM' 18)	netmf
ProNE (Zhang et al, IJCAI' 19)	prone
NetSMF (Qiu et at, WWW' 19)	netsmf
Node2vec (Grover et al, KDD' 16)	node2vec
LINE (Tang et al, WWW' 15)	line
DeepWalk (Perozzi et al, KDD' 14)	deepwalk
Spectral (Tang et al, Data Min Knowl Disc (2011))	spectral
Hope (Ou et al, KDD' 16)	hope
GraRep (Cao et al, CIKM' 15)	grarep

Semi-Supervised Node Classification with Attributes

Model	Name in Cogdl
Grand(Feng et al.,NLPS' 20)	grand
GCNII(Chen et al.,ICML' 20)	gcnii
DR-GAT (Zou et al., 2019)	drgat
MVGRL (Hassani et al., KDD' 20)	mvgrl
APPNP (Klicpera et al., ICLR' 19)	ppnp
GAT (Veličković et al., ICLR' 18)	gat
GDC_GCN (Klicpera et al., NeurIPS' 19)	gdc_gcn
DropEdge (Rong et al., ICLR' 20)	dropedge_gcn
GCN (Kipf et al., ICLR' 17)	gcn
DGI (Veličković et al., ICLR' 19)	dgi
GraphSAGE (Hamilton et al., NeurIPS' 17)	graphsage
GraphSAGE (unsup)(Hamilton et al., NeurIPS' 17)	unsup_graphsage
MixHop (Abu-El-Haija et al., ICML' 19)	mixhop

Multiplex Node Classification

Model	Name in Cogdl
Simple-HGN (Lv and Ding et al, KDD' 21)	simple-hgn
GTN (Yun et al, NeurIPS' 19)	gtn
HAN (Xiao et al, WWW' 19)	han
GCC (Qiu et al, KDD' 20)	gcc
PTE (Tang et al, KDD' 15)	pte
Metapath2vec (Dong et al, KDD' 17)	metapath2vec
Hin2vec (Fu et al, CIKM' 17)	hin2vec

Link Prediction

Model	Name in Cogdl
ProNE (Zhang et al, IJCAI'19)	prone
NetMF (Qiu et al, WSDM'18)	netmf
Hope (Ou et al, KDD'16)	hope
LINE (Tang et al, WWW'15)	line
Node2vec (Grover et al, KDD'16)	node2vec
NetSMF (Qiu et at, WWW'19)	netsmf
DeepWalk (Perozzi et al, KDD'14)	deepwalk
SDNE (Wang et al, KDD'16)	sdne

Multiplex Link Prediction

Model	Name in Cogdl
GATNE (Cen et al, KDD'19)	gatne
NetMF (Qiu et al, WSDM'18)	netmf
ProNE (Zhang et al, IJCAI'19)	prone++
Node2vec (Grover et al, KDD'16)	node2vec
DeepWalk (Perozzi et al, KDD'14)	deepwalk
LINE (Tang et al, WWW'15)	line
Hope (Ou et al, KDD'16)	hope
GraRep (Cao et al, CIKM'15)	grarep

Knowledge graph completion

Model	Name in Cogdl
CompGCN (Vashishth et al, ICLR'20)	compgcn

Graph Classification

Model	Name in Cogdl
GIN (Xu et al, ICLR' 19)	gin
Infograph (Sun et al, ICLR' 20)	infograph
DiffPool (Ying et al, NeuIPS' 18)	diffpool
SortPool (Zhang et al, AAAI' 18)	softpool
Graph2Vec (Narayanan et al, CoRR' 17)	graph2vec
PATCH_SAN (Niepert et al, ICML' 16)	patchy_san
DGK (Yanardag et al, KDD' 15)	dgk

Attributed graph clustering

Model	Name in Cogdl
AGC (Zhang et al, IJCAI 19)	agc
DAEGC (Wang et al, ICLR' 20)	daegc

2.5 Model Training

2.5.1 Customized model training logic

cogdl supports the selection of custom training logic, you can use the models and data sets in Cogdl to achieve your personalized needs.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from cogdl import experiment
from cogdl.datasets import build_dataset_from_name
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel
class Gnn(BaseModel):
    def __init__(self, in_feats, hidden_size, out_feats, dropout):
        super(Gnn, self).__init__()
        self.conv1 = GCNLayer(in_feats, hidden_size)
        self.conv2 = GCNLayer(hidden_size, out_feats)
        self.dropout = nn.Dropout(dropout)
    def forward(self, graph):
        graph.sym_norm()
```

(continues on next page)

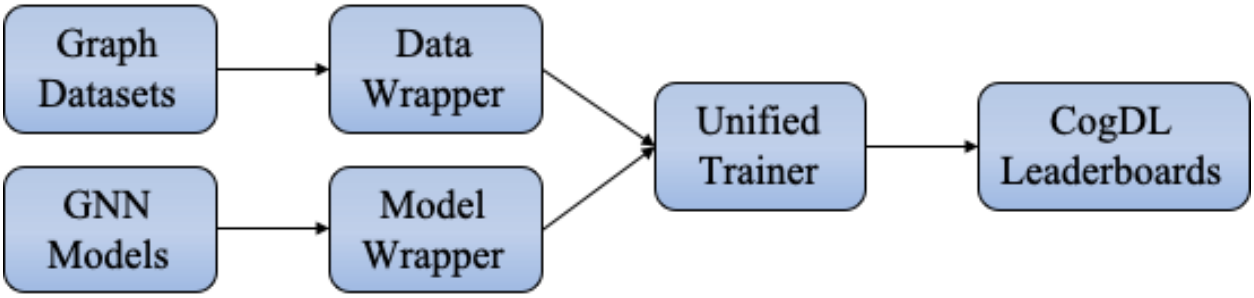
(continued from previous page)

```
        h = graph.x
        h = F.relu(self.conv1(graph, self.dropout(h)))
        h = self.conv2(graph, self.dropout(h))
        return F.log_softmax(h, dim=1)

if __name__ == "__main__":
    dataset = build_dataset_from_name("cora")[0]
    model = Gnn(in_feats=dataset.num_features, hidden_size=64, out_feats=dataset.num_
↪classes, dropout=0.1)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-3)
    model.train()
    for epoch in range(300):
        optimizer.zero_grad()
        out = model(dataset)
        loss = F.nll_loss(out[dataset.train_mask], dataset.y[dataset.train_mask])
        loss.backward()
        optimizer.step()
    model.eval()
    _, pred = model(dataset).max(dim=1)
    correct = float(pred[dataset.test_mask].eq(dataset.y[dataset.test_mask]).sum().
↪item())
    acc = correct / dataset.test_mask.sum().item()
    print('The accuracy rate obtained by running the experiment with the custom␣
↪training logic: {:.6f}'.format(acc))
```

2.5.2 Unified Trainer

CogDL provides a unified trainer for GNN models, which takes over the entire loop of the training process. The unified trainer, which contains much engineering code, is implemented flexibly to cover arbitrary GNN training settings.



We design four decoupled modules for the GNN training, including *Model*, *Model Wrapper*, *Dataset*, *Data Wrapper*. The *Model Wrapper* is for the training and testing steps, while the *Data Wrapper* is designed to construct data loaders used by *Model Wrapper*.

The main contributions of most GNN papers mainly lie on three modules except *Dataset*, as shown in the table. For

example, the GCN paper trains the GCN model under the (semi-)supervised and full-graph setting, while the DGI paper trains the GCN model by maximizing local-global mutual information. The training method of the DGI is considered as a model wrapper named *dgi_mw*, which could be used for other scenarios.

Paper	Model	Model Wrapper	Data Wrapper
GCN	GCN	supervised	full-graph
GAT	GAT	supervised	full-graph
GraphSAGE	SAGE	sage_mw	neighbor sampling
Cluster-GCN	GCN	supervised	graph clustering
DGI	GCN	dgi_mw	full-graph

Based on the design of the unified trainer and decoupled modules, we could do arbitrary combinations of models, model wrappers, and data wrappers. For example, if we want to apply DGI to large-scale datasets, all we need is to substitute the full-graph data wrapper with the neighbor-sampling or clustering data wrappers without additional modifications. If we propose a new GNN model, all we need is to write essential PyTorch-style code for the model. The rest could be automatically handled by CogDL by specifying the model wrapper and the data wrapper. We could quickly conduct experiments for the model using the trainer via `trainer = Trainer(epochs,...)` and `trainer.run(...)`. Moreover, based on the unified trainer, CogDL provides native support for many useful features, including hyperparameter optimization, efficient training techniques, and experiment management without any modification to the model implementation.

2.5.3 Experiment API

CogDL provides a more easy-to-use API upon *Trainer*, i.e., *experiment*. 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. 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")
```

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

```
from cogdl import experiment
from cogdl.datasets import build_dataset
from cogdl.models import build_model
from cogdl.options import get_default_args

args = get_default_args(model="gcn", dataset="cora")
dataset = build_dataset(args)
```

(continues on next page)

(continued from previous page)

```
model = build_model(args)
experiment(model=model, dataset=dataset)
```

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

2.5.4 How to save trained model?

CogDL supports saving the trained model with `checkpoint_path` in command line or API usage. For example:

```
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt")
```

When the training stops, the model will be saved in *gcn_cora.pt*. 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", dataset="cora", checkpoint_path="gcn_cora.pt", resume_
↪training=True)
```

In command line usage, the same results can be achieved with `--checkpoint-path {path}` and `--resume-training`.

2.5.5 How to 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 the given path specified by `--save-emb-path {path}`.

```
experiment(model="prone", dataset="blogcatalog", save_emb_path="./embeddings/prone_
↪blog.npy")
```

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",
    load_emb_path="./embeddings/prone_blog.npy",
    num_shuffle=5,
    training_percents=[0.1, 0.5, 0.9]
)
```

You can also use command line to achieve the same results

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

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

2.6 Using Customized Dataset

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

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

2.6.1 Dataset for node_classification

To create a dataset for node_classification, you need to inherit `NodeDataset`. `NodeDataset` is for 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, generate_random_graph

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

    def process(self):
        """You need to load your dataset and transform to `Graph`"""
        num_nodes, num_edges, feat_dim = 100, 300, 30

        # load or generate your dataset
        edge_index = torch.randint(0, num_nodes, (2, num_edges))
        x = torch.randn(num_nodes, feat_dim)
        y = torch.randint(0, 2, (num_nodes,))

        # set train/val/test mask in node_classification task
        train_mask = torch.zeros(num_nodes).bool()
        train_mask[0 : int(0.3 * num_nodes)] = True
        val_mask = torch.zeros(num_nodes).bool()
        val_mask[int(0.3 * num_nodes) : int(0.7 * num_nodes)] = True
        test_mask = torch.zeros(num_nodes).bool()
        test_mask[int(0.7 * num_nodes) :] = True
        data = Graph(x=x, edge_index=edge_index, y=y, train_mask=train_mask, val_
↪mask=val_mask, test_mask=test_mask)
        return data

if __name__ == "__main__":
    # Train customized dataset via defining a new class
    dataset = MyNodeDataset()
    experiment(dataset=dataset, model="gcn")

    # Train customized dataset via feeding the graph data to NodeDataset
    data = generate_random_graph(num_nodes=100, num_edges=300, num_feats=30)
    dataset = NodeDataset(data=data)
    experiment(dataset=dataset, model="gcn")
```

2.6.2 Dataset for graph_classification

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

```
from cogdl.data import Graph
from cogdl.datasets import GraphDataset

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))
        return graphs

if __name__ == "__main__":
    dataset = MyGraphDataset()
    experiment(model="gin", dataset=dataset)
```

2.7 Using Customized GNN

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

2.7.1 GNN layers in CogDL to Define model

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

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

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

```
import torch
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel

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

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

2.7.2 Define your GNN Module

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

```
import torch
from cogdl.utils import spmm

class GCNLayer(torch.nn.Module):
    """
    Args:
        in_feats: int
```

(continues on next page)

(continued from previous page)

```
            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):
        h = self.fc(x)
        h = spmm(graph, h)
        return h
```

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

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

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

2.7.3 Use Custom models with CogDL

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

```
data = build_dataset_from_name("cora")[0]
# Use the JKNet model as defined above
model = JKNet(data.num_features, data.num_classes, 32, 4)
experiment(model=model, dataset="cora", mw="node_classification_mw", dw="node_
↪classification_dw")
```

2.8 Code Gallery

Below is a code of examples

2.8.1 Introduce of Graphs

How to represent a graph in CogDL

```
import torch
from cogdl.data import Graph
edges = torch.tensor([[0,1],[1,3],[2,1],[4,2],[0,3]]).t()
x = torch.tensor([[-1],[0],[1],[2],[3]])
g = Graph(edge_index=edges,x=x) # equivalent to that above
print(g.row_indptr)

print(g.col_indices)

print(g.edge_weight)

print(g.num_nodes)

print(g.num_edges)

g.edge_weight = torch.rand(5)
print(g.edge_weight)
```

How to construct 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")

print(dataset[0])

loader = DataLoader(dataset, batch_size=8)
for batch in loader:
    model(batch)
```

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

    return out
```

How to edit the graph?

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]
print(graph.num_edges)

with graph.local_graph():
    mask = torch.arange(100)
    row, col = graph.edge_index
    graph.edge_index = (row[mask], col[mask])
    print(graph.num_edges)

print(graph.num_edges)


print(graph.edge_weight)

with graph.local_graph():
    graph.edge_weight += 1
print(graph.edge_weight)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.8.2 Model Training

Customized model training logic

cogdl supports the selection of custom training logic, you can use the models and data sets in Cogdl to achieve your personalized needs.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from cogdl import experiment
from cogdl.datasets import build_dataset_from_name
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel
class Gnn(BaseModel):
    def __init__(self, in_feats, hidden_size, out_feats, dropout):
        super(Gnn, self).__init__()
        self.conv1 = GCNLayer(in_feats, hidden_size)
        self.conv2 = GCNLayer(hidden_size, out_feats)
        self.dropout = nn.Dropout(dropout)
    def forward(self, graph):
        graph.sym_norm()
        h = graph.x
        h = F.relu(self.conv1(graph, self.dropout(h)))
        h = self.conv2(graph, self.dropout(h))
        return F.log_softmax(h, dim=1)

if __name__ == "__main__":
    dataset = build_dataset_from_name("cora")[0]
    model = Gnn(in_feats=dataset.num_features, hidden_size=64, out_feats=dataset.num_
↪classes, dropout=0.1)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-3)
    model.train()
    for epoch in range(300):
        optimizer.zero_grad()
        out = model(dataset)
        loss = F.nll_loss(out[dataset.train_mask], dataset.y[dataset.train_mask])
        loss.backward()
        optimizer.step()
    model.eval()
    _, pred = model(dataset).max(dim=1)
    correct = float(pred[dataset.test_mask].eq(dataset.y[dataset.test_mask]).sum().
↪item())
    acc = correct / dataset.test_mask.sum().item()
```

(continues on next page)

(continued from previous page)

```
    print('The accuracy rate obtained by running the experiment with the custom␣
↪training logic: {:.6f}'.format(acc))
```

Experiment API

CogDL provides a more easy-to-use API upon Trainer, i.e., experiment

```
from cogdl import experiment
experiment(model="gcn", dataset="cora")
# Or you can create each component separately and manually run the process using␣
↪build_dataset, build_model in CogDL.

from cogdl import experiment
from cogdl.datasets import build_dataset
from cogdl.models import build_model
from cogdl.options import get_default_args

args = get_default_args(model="gcn", dataset="cora")
dataset = build_dataset(args)
model = build_model(args)
experiment(model=model, dataset=dataset)
```

How to save trained model?

```
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt")

# When the training stops, the model will be saved in gcn_cora.pt. 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", dataset="cora", checkpoint_path="gcn_cora.pt", resume_
↪training=True)
```

How to save embeddings?

```
experiment(model="prone", dataset="blogcatalog", save_emb_path="./embeddings/prone_
↪blog.npy")

# 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",
    load_emb_path="./embeddings/prone_blog.npy",
    num_shuffle=5,
    training_percents=[0.1, 0.5, 0.9]
)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.8.3 Using Customized Dataset

Dataset for node_classification

```
import torch
from cogdl import experiment
from cogdl.data import Graph
from cogdl.datasets import NodeDataset, generate_random_graph

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

    def process(self):
        """You need to load your dataset and transform to `Graph`"""
        num_nodes, num_edges, feat_dim = 100, 300, 30

        # load or generate your dataset
        edge_index = torch.randint(0, num_nodes, (2, num_edges))
        x = torch.randn(num_nodes, feat_dim)
        y = torch.randint(0, 2, (num_nodes,))

        # set train/val/test mask in node_classification task
```

(continues on next page)

(continued from previous page)

```
        train_mask = torch.zeros(num_nodes).bool()
        train_mask[0 : int(0.3 * num_nodes)] = True
        val_mask = torch.zeros(num_nodes).bool()
        val_mask[int(0.3 * num_nodes) : int(0.7 * num_nodes)] = True
        test_mask = torch.zeros(num_nodes).bool()
        test_mask[int(0.7 * num_nodes) :] = True
        data = Graph(x=x, edge_index=edge_index, y=y, train_mask=train_mask, val_
↪mask=val_mask, test_mask=test_mask)
        return data

if __name__ == "__main__":
    # Train customized dataset via defining a new class
    dataset = MyNodeDataset()
    experiment(dataset=dataset, model="gcn")

    # Train customized dataset via feeding the graph data to NodeDataset
    data = generate_random_graph(num_nodes=100, num_edges=300, num_feats=30)
    dataset = NodeDataset(data=data)
    experiment(dataset=dataset, model="gcn")
```

Dataset for graph_classification

```
from cogdl.data import Graph
from cogdl.datasets import GraphDataset

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))
        return graphs

if __name__ == "__main__":
    dataset = MyGraphDataset()
```

(continues on next page)

(continued from previous page)

```
    experiment(model="gin", dataset=dataset)
```

Total running time of the script: (0 minutes 0.000 seconds)

2.8.4 Using Customized GNN

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:

```
import torch
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel

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

    def forward(self, graph):
        # symmetric normalization of adjacency matrix
        graph.sym_norm()
        h = graph.x
        out = []
        for layer in self.layers:
            h = layer(graph,h)
            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):
        h = self.fc(x)
        h = spmm(graph, h)
        return h
```

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 = build_dataset_from_name("cora")[0]
# Use the JKNet model as defined above
model = JKNet(data.num_features, data.num_classes, 32, 4)
experiment(model=model, dataset="cora", mw="node_classification_mw", dw="node_
↪classification_dw")
```

Total running time of the script: (0 minutes 0.000 seconds)

2.9 中文教程

2.9.1 安装

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

请按照此处的说明安装 PyTorch

安装 PyTorch 后，可以使用 pip 命令安装 cogdl，如下所示：

```
pip install cogdl
```

或者 Install from source via：

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

或者 clone 仓库并使用以下命令进行安装：

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

如果您想使用 PyTorch Geometric (PyG) 中的模块，您可以按照此处的说明安装 PyTorch Geometric

2.9.2 快速开始

API 用法

您可以通过 CogDL 的 API 运行各种实验，尤其是 experiment(). 您还可以使用自己的数据集和模型进行实验。快速入门的示例可以在 quick_start.py. 中找到。examples/ 中提供了更多的示例。

```
from cogdl import experiment

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

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

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

# automl usage
```

(continues on next page)

(continued from previous page)

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

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

命令行用法

您可以使用命令 `python scripts/train.py --dataset example_dataset --model example_model` 运行 example_model 在 example_data 上.

- `--dataset`, 是要使用的数据集名称, 可以是带空格的数据集列表比如 `cora citeseer`. 支持的数据集包括 `cora`, `citeseer`, `pumbed`, `ppi`, `flickr` 等等. 查看更多的数据集 cogdl/datasets
- `--model`, 是要使用的模型名称, 可以是带空格的数据集列表比如 `gcn gat`. 支持的模型包括 `gcn`, `gat`, `graphsage` 等等. 查看更多的模型 cogdl/models.

例如，如果你想在 Cora 数据集上运行 GCN 和 GAT，使用 5 个不同的 seeds：

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

预期结果:

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

如果您想在多个模型/数据集上使用多个 GPU 在您的服务器上并行的进行实验：

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

预期输出:

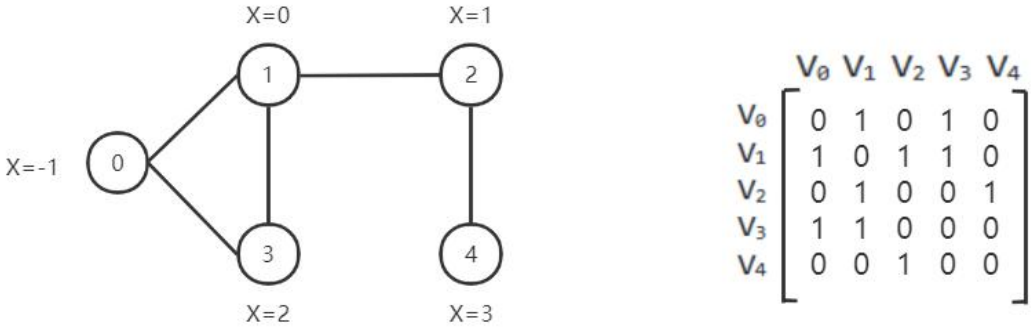
Variant	test_acc	val_acc
（‘cora’，‘gcn’）	0.8050±0.0047	0.7940±0.0063
（‘cora’，‘gat’）	0.8234±0.0042	0.8088±0.0016
（‘citeseer’，‘gcn’）	0.6938±0.0133	0.7108±0.0148
（‘citeseer’，‘gat’）	0.7098±0.0053	0.7244±0.0039

2.9.3 图简介

真实世界的图表

图结构数据已在许多现实世界场景中得到广泛应用。例如，Facebook 上的每个用户都可以被视为一个顶点，而他们之间比如友谊或追随性等关系可以被视为图中的边。我们可能对预测用户的兴趣或者对网络中的一对节点是否有有连接它们的边感兴趣。

我们可以使用邻接矩阵表示一张图



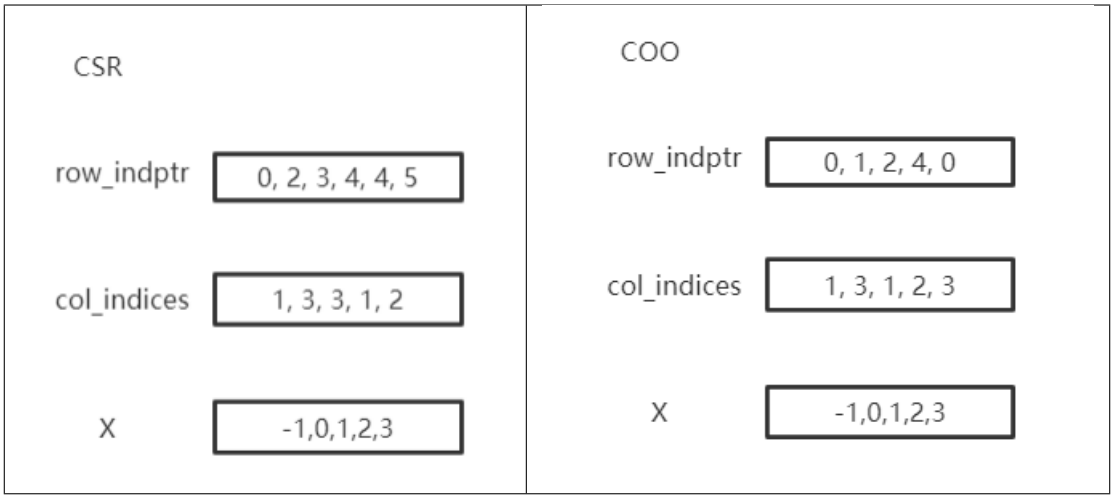
如何在 CogDL 中表示图形

图用于存储结构化数据的信息, 在 CogDL 中使用 cogdl.data.Graph 对象来表示一张图。简而言之，一个 Graph 具有以下属性：

- x: 节点特征矩阵，shape[num_nodes, num_features]，torch.Tensor
- edge_index：COO 格式的稀疏矩阵，tuple
- edge_weight：边权重 shape[num_edges,]，torch.Tensor
- edge_attr：边属性矩阵 shape[num_edges, num_attr]
- y: 每个节点的目标标签, 单标签情况下 shape [num_nodes,], 多标签情况下的 shape [num_nodes, num_labels]
- row_indptr：CSR 稀疏矩阵的行索引指针，torch.Tensor。
- col_indices：CSR 稀疏矩阵的列索引，torch.Tensor。
- num_nodes：图中的节点数。
- num_edges：图中的边数。

以上是基本属性，但不是必需的。你可以用 g = Graph(edge_index=edges) 定义一个图并省略其他属性。此外，Graph 不限于这些属性，还支持其他自定义属性，例如 graph.mask = mask。

在 Cogdl 中表示这张图



Graph 以 COO 或 CSR 格式存储稀疏矩阵。COO 格式更容易添加或删除边，例如 add_self_loops，使用 CSR 存储是为了用于快速消息传递。Graph 自动在两种格式之间转换，您可以按需使用两种格式而无需担心。您可以创建带有边的图形或将边分配给创建的图形。edge_weight 将自动初始化为 1，您可以根据需要对其进行修改。

```
import torch
from cogdl.data import Graph
edges = torch.tensor([[0,1],[1,3],[2,1],[4,2],[0,3]]).t()
x = torch.tensor([[-1],[0],[1],[2],[3]])
g = Graph(edge_index=edges,x=x) # equivalent to that above
print(g.row_indptr)
>>tensor([0, 2, 3, 4, 4, 5])
print(g.col_indices)
>>tensor([1, 3, 3, 1, 2])
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])
```

我们在 Graph 中实现了常用的操作：

- `add_self_loops`: 为图中的节点添加自循环

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

- `add_remaining_self_loops`: 为图中还没有自环的节点添加自环

- `sym_norm`: 使用 GCN 的 edge_weight 的对称归一化

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

- `row_norm`: edge_weight 的逐行归一化:

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

- `degrees`: 获取每个节点的度数。对于有向图，此函数返回每个节点的入度

```
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`: 得到一个包含给定节点和它们之间的边的子图。
- `edge_subgraph`: 得到一个包含给定边和相应节点的子图。
- `sample_adj`: 为每个给定节点采样固定数量的邻居

```
from cogdl.datasets import build_dataset_from_name
g = build_dataset_from_name("cora")[0]
g.num_nodes
>> 2708
g.num_edges
>> 10556
# 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`: 在 inductive 的设置中, 一些节点和边在 trainning 中看不见的, 对于 training/evaluation 使用 `train/eval` 来切换 backend graph. 在 transductive 设置中, 您可以忽略这一点.

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

# inference_step
```

(continues on next page)

(continued from previous page)

```
model.eval()
graph.eval()
```

如何构建 mini-batch graphs

在节点分类中，所有操作都在一个图中。但是在像图分类这样的任务中，我们需要用 mini-batch 处理很多图。图分类的数据集包含可以使用索引访问的图，例如 data [2]。为了支持小批量训练/推理，CogDL 将一批中的图组合成一个完整的图，其中邻接矩阵形成稀疏块对角矩阵，其他的（节点特征、标签）在节点维度上连接。这个过程由由 cogdl.data.Dataloader 来处理。

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

dataset = build_dataset_from_name("mutag")
>> MUTAGDataset(188)
dataset[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 是一个附加属性，指示节点所属的各个图。它主要用于做全局池化，或者称为 readout 来生成 graph-level 表示。具体来说，batch 是一个像这样的张量

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

以下代码片段显示了如何进行全局池化对每个图中节点的特征进行求和

```
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
```

如何编辑一个 graph?

在某些设置中，可以更改 edges. 在这种情况下，我们需要在保留原始图的同时生成计算图。CogDL 提供了 graph.local_graph 来设置 local scape，任何 out-of-place 操作都不会反映到原始图上。但是，in-place 操作会影响原始图形。

```
graph = build_dataset_from_name("cora")[0]
graph.num_edges
>> 10556
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
>> 10556

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

常见的 graph 数据集

CogDL 为节点分类、图分类等任务提供了一些常用的数据集。您可以方便地访问它们，如下所示：

```
from cogdl.datasets import build_dataset_from_name
dataset = build_dataset_from_name("cora")

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

对于节点分类的所有数据集，我们使用 train_mask、val_mask、test_mask 来表示节点的训练/验证/测试拆分。

CogDL 现在支持以下的数据集用于不同的任务：

- Network Embedding (无监督节点分类): PPI, Blogcatalog, Wikipedia, Youtube, DBLP, Flickr
- 半监督/无监督节点分类: Cora, Citeseer, Pubmed, Reddit, PPI, PPI-large, Yelp, Flickr, Amazon
- 异构节点分类: DBLP, ACM, IMDB
- 链接预测: PPI, Wikipedia, Blogcatalog

- 多路链接预测: Amazon, YouTube, Twitter
- 图分类: MUTAG, IMDB-B, IMDB-M, PROTEINS, COLLAB, NCI, NCI109, Reddit-BINARY

Network Embedding(无监督节点分类)

Dataset	Nodes	Edges	Classes	Degree	Name in Cogdl
PPI	3,890	76,584	50(m)	—	ppi-ne
BlogCatalog	10,312	333,983	40(m)	32	blogcatalog
Wikipedia	4.777	184,812	39(m)	39	wikipedia
Flickr	80,513	5,899,882	195(m)	73	flickr-ne
DBLP	51,264	2,990,443	60(m)	2	dblp-ne
Youtube	1,138,499	2,990,443	47(m)	3	youtube-ne

节点分类

Dataset	Nodes	Edges	Features	Classes	Train/Val/Test	Degree	Name in cogdl
Cora	2,708	5,429	1,433	7(s)	140 / 500 / 1000	2	cora
Citeseer	3,327	4,732	3,703	6(s)	120 / 500 / 1000	1	citeseer
PubMed	19,717	44,338	500	3(s)	60 / 500 / 1999	2	pubmed
Chameleon	2,277	36,101	2,325	5	0.48 / 0.32 / 0.20	16	chameleon
Cornell	183	298	1,703	5	0.48 / 0.32 / 0.20	1.6	cornell
Film	7,600	30,019	932	5	0.48 / 0.32 / 0.20	4	film
Squirrel	5201	217,073	2,089	5	0.48 / 0.32 / 0.20	41.7	squirrel
Texas	182	325	1,703	5	0.48 / 0.32 / 0.20	1.8	texas
Wisconsin	251	515	1,703	5	0.48 / 0.32 / 0.20	2	Wisconsin
PPI	14,755	225,270	50	121(m)	0.66 / 0.12 / 0.22	15	ppi
PPI-large	56,944	818,736	50	121(m)	0.79 / 0.11 / 0.10	14	ppi-large
Reddit	232,965	11,606,919	602	41(s)	0.66 / 0.10 / 0.24	50	reddit
Flickr	89,250	899,756	500	7(s)	0.50 / 0.25 / 0.25	10	flickr
Yelp	716,847	6,977,410	300	100(m)	0.75 / 0.10 / 0.15	10	yelp
Amazon-SAINT	1,598,960	132,169,734	200	107(m)	0.85 / 0.05 / 0.10	83	amazon-s

异构图

Dataset	Nodes	Edges	Features	Classes	Train/Val/Test	Degree	Edge Type	Name in Cogdl
DBLP	18,405	67,946	334	4	800 / 400 / 2857	4	4	gtn-dblp(han-acm)
ACM	8,994	25,922	1,902	3	600 / 300 / 2125	3	4	gtn-acm(han-acm)
IMDB	12,772	37,288	1,256	3	300 / 300 / 2339	3	4	gtn-imdb(han-imdb)
Amazon-GATNE	10,166	148,863	—	—	—	15	2	amazon
Youtube-GATNE	2,000	1,310,617	—	—	—	655	5	youtube
Twitter	10,000	331,899	—	—	—	33	4	twitter

知识图谱链接预测

Dataset	Nodes	Edges	Train/Val/Test	Relations Types	Degree	Name in Cogdl
FB13	75,043	345,872	316,232 / 5,908 / 23,733	12	5	fb13
FB15k	14,951	592,213	483,142 / 50,000 / 59,071	1345	40	fb15k
FB15k-237	14,541	310,116	272,115 / 17,535 / 20,466	237	21	fb15k237
WN18	40,943	151,442	141,442 / 5,000 / 5,000	18	4	wn18
WN18RR	86,835	93,003	86,835 / 3,034 / 3,134	11	1	wn18rr

图分类

TUdataset from https://www.chrsmrrs.com/graphkerneldatasets

Dataset	Graphs	Classes	Avg. Size	Name in Cogdl
MUTAG	188	2	17.9	mutag
IMDB-B	1,000	2	19.8	imdb-b
IMDB-M	1,500	3	13	imdb-m
PROTEINS	1,113	2	39.1	proteins
COLLAB	5,000	5	508.5	collab
NCI1	4,110	2	29.8	nci1
NCI109	4,127	2	39.7	nci109
PTC-MR	344	2	14.3	ptc-mr
REDDIT-BINARY	2,000	2	429.7	reddit-b
REDDIT-MULTI-5k	4,999	5	508.5	reddit-multi-5k
REDDIT-MULTI-12k	11,929	11	391.5	reddit-multi-12k
BBBP	2,039	2	24	bbbp
BACE	1,513	2	34.1	bace

2.9.4 Cogdl 中的模型

图表示学习介绍

受最近计算机视觉和自然语言处理方面的表示学习趋势的启发，图表示学习被提出。图表示旨在学习顶点/图的低维连续向量，同时保留内在图属性，或者使用图编码器进行端到端训练。最近，已经提出了图神经网络（GNN），并在半监督表示学习中取得了令人印象深刻的性能。图卷积网络 (GCN) 通过谱图卷积的局部一阶近似提出了一种卷积架构。Gra phSAGE 是一个通用归纳框架，它利用节点特征为以前未见过的样本生成节点 embeddings。Graph Attention Networks (GATs) 利用多头自注意力机制，并能够（隐式）为邻域中的不同节点指定不同的权重。

CogDL 现在支持以下任务

- unsupervised node classification(无监督节点分类)
- semi-supervised node classification(半监督节点分类)
- heterogeneous node classification(异构节点分类)
- link prediction(链接预测)
- multiplex link prediction(多路链接预测)
- unsupervised graph classification(无监督图分类)
- supervised graph classification(监督图分类)
- graph pre-training(图预训练)
- attributed graph clustering(属性图聚类)

CogDL 提供了丰富的通用基准数据集和 GNN 模型。您可以使用 CogDL 中的模型和数据集简单地开始运行。

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

Unsupervised Multi-label Node Classification

Model	Name in Cogdl
NetMF (Qiu et al, WSDM'18)	netmf
ProNE (Zhang et al, IJCAI'19)	prone
NetSMF (Qiu et at, WWW'19)	netsmf
Node2vec (Grover et al, KDD'16)	node2vec
LINE (Tang et al, WWW'15)	line
DeepWalk (Perozzi et al, KDD'14)	deepwalk
Spectral (Tang et al, Data Min Knowl Disc (2011))	spectral
Hope (Ou et al, KDD'16)	hope
GraRep (Cao et al, CIKM'15)	grarep

Semi-Supervised Node Classification with Attributes

Model	Name in Cogdl
Grand(Feng et al.,NLPS'20)	grand
GCNII((Chen et al.,ICML'20)	gcnii
DR-GAT (Zou et al., 2019)	drgat
MVGRL (Hassani et al., KDD'20)	mvgrl
APPNP (Klicpera et al., ICLR'19)	ppnp
GAT (Veličković et al., ICLR'18)	gat
GDC_GCN (Klicpera et al., NeurIPS'19)	gdc_gcn
DropEdge (Rong et al., ICLR'20)	dropedge_gcn
GCN (Kipf et al., ICLR'17)	gcn
DGI (Veličković et al., ICLR'19)	dgi
GraphSAGE (Hamilton et al., NeurIPS'17)	graphsage
GraphSAGE (unsup)(Hamilton et al., NeurIPS'17)	unsup_graphsage
MixHop (Abu-El-Haija et al., ICML'19)	mixhop

Multiplex Node Classification

Model	Name in Cogdl
Simple-HGN (Lv and Ding et al, KDD' 21)	simple-hgn
GTN (Yun et al, NeurIPS' 19)	gtn
HAN (Xiao et al, WWW' 19)	han
GCC (Qiu et al, KDD' 20)	gcc
PTE (Tang et al, KDD' 15)	pte
Metapath2vec (Dong et al, KDD' 17)	metapath2vec
Hin2vec (Fu et al, CIKM' 17)	hin2vec

Link Prediction

Model	Name in Cogdl
ProNE (Zhang et al, IJCAI' 19)	prone
NetMF (Qiu et al, WSDM' 18)	netmf
Hope (Ou et al, KDD' 16)	hope
LINE (Tang et al, WWW' 15)	line
Node2vec (Grover et al, KDD' 16)	node2vec
NetSMF (Qiu et at, WWW' 19)	netsmf
DeepWalk (Perozzi et al, KDD' 14)	deepwalk
SDNE (Wang et al, KDD' 16)	sdne

Multiplex Link Prediction

Model	Name in Cogdl
GATNE (Cen et al, KDD' 19)	gatne
NetMF (Qiu et al, WSDM' 18)	netmf
ProNE (Zhang et al, IJCAI' 19)	prone++
Node2vec (Grover et al, KDD' 16)	node2vec
DeepWalk (Perozzi et al, KDD' 14)	deepwalk
LINE (Tang et al, WWW' 15)	line
Hope (Ou et al, KDD' 16)	hope
GraRep (Cao et al, CIKM' 15)	grarep

Knowledge graph completion

Model	Name in Cogdl
CompGCN (Vashishth et al, ICLR' 20)	compgcn

Graph Classification

Model	Name in Cogdl
GIN (Xu et al, ICLR' 19)	gin
Infograph (Sun et al, ICLR' 20)	infograph
DiffPool (Ying et al, NeuIPS' 18)	diffpool
SortPool (Zhang et al, AAAI' 18)	softpool
Graph2Vec (Narayanan et al, CoRR' 17)	graph2vec
PATCH_SAN (Niepert et al, ICML' 16)	patchy_san
DGK (Yanardag et al, KDD' 15)	dgk

Attributed graph clustering

Model	Name in Cogdl
AGC (Zhang et al, IJCAI 19)	agc
DAEGC (Wang et al, ICLR' 20)	daegc

2.9.5 模型训练

自定义模型训练逻辑

cogdl 支持选择自定义训练逻辑，"数据-模型-训练"三部分在 CogDL 中是独立的，研究者和使用者可以自定义其中任何一部分，并复用其他部分，从而提高开发效率。现在您可以使用 Cogdl 中的模型和数据集来实现您的个性化需求。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from cogdl import experiment
from cogdl.datasets import build_dataset_from_name
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel
class Gnn(BaseModel):
```

(continues on next page)

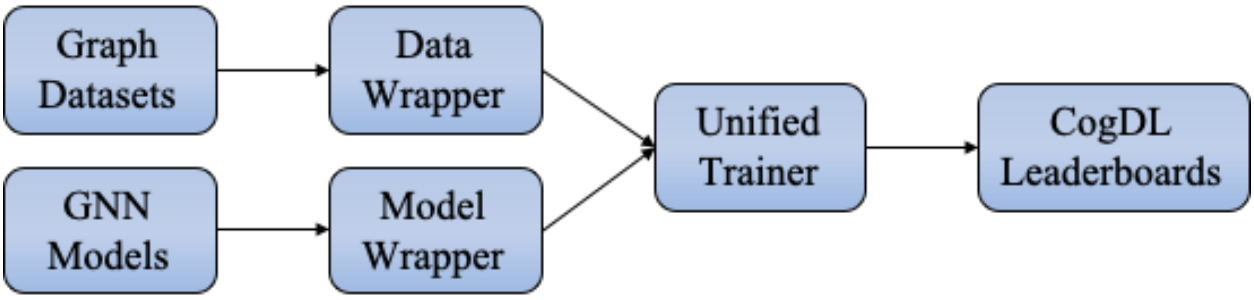
(continued from previous page)

```
    def __init__(self, in_feats, hidden_size, out_feats, dropout):
        super(Gnn, self).__init__()
        self.conv1 = GCNLayer(in_feats, hidden_size)
        self.conv2 = GCNLayer(hidden_size, out_feats)
        self.dropout = nn.Dropout(dropout)
    def forward(self, graph):
        graph.sym_norm()
        h = graph.x
        h = F.relu(self.conv1(graph, self.dropout(h)))
        h = self.conv2(graph, self.dropout(h))
        return F.log_softmax(h, dim=1)

if __name__ == "__main__":
    dataset = build_dataset_from_name("cora")[0]
    model = Gnn(in_feats=dataset.num_features, hidden_size=64, out_feats=dataset.num_
↪classes, dropout=0.1)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-3)
    model.train()
    for epoch in range(300):
        optimizer.zero_grad()
        out = model(dataset)
        loss = F.nll_loss(out[dataset.train_mask], dataset.y[dataset.train_mask])
        loss.backward()
        optimizer.step()
    model.eval()
    _, pred = model(dataset).max(dim=1)
    correct = float(pred[dataset.test_mask].eq(dataset.y[dataset.test_mask]).sum().
↪item())
    acc = correct / dataset.test_mask.sum().item()
    print('The accuracy rate obtained by running the experiment with the custom␣
↪training logic: {:.6f}'.format(acc))
```

统一训练器

CogDL 为 GNN 模型提供了一个统一的训练器，它接管了训练过程的整个循环。包含大量工程代码的统一训练器可灵活实现以涵盖任意 GNN 训练设置



为了更方便的使用 GNN 训练，我们设计了四个解耦模块，包括 Model、Model Wrapper、Dataset、Data Wrapper。Model Wrapper 用于训练和测试步骤，而 Data Wrapper 旨在构建 Model Wrapper 使用的数据加载器。大多数 GNN 论文的主要贡献主要在于除 Dataset 之外的三个模块，如下表所示。例如，GCN 论文在（半）监督和全图设置下训练 GCN 模型，而 DGI 论文通过最大化局部-全局互信息来训练 GCN 模型。DGI 的训练方法被认为是一个 dgi_mw 的模型包装器，可以用于其他场景。

Paper	Model	Model Wrapper	Data Wrapper
GCN	GCN	supervised	full-graph
GAT	GAT	supervised	full-graph
GraphSAGE	SAGE	sage_mw	neighbor sampling
Cluster-GCN	GCN	supervised	graph clustering
DGI	GCN	dgi_mw	full-graph

基于统一训练器和解耦模块的设计，我们可以对模型、Model Wrapper 和 Data Wrapper 进行任意组合。例如，如果我们想将 DGI 应用于大规模数据集，我们只需要用邻居采样或聚类数据包装器替换全图 data wrapper，而无需额外修改。如果我们提出一个新的 GNN 模型，我们只需要为模型编写必要的 PyTorch 风格的代码。其余的可以通过指定 Model Wrapper 和 Data Wrapper 由 CogDL 自动处理。我们可以通过 `trainer = Trainer(epochs,...)` 和 `trainer.run(...)`. 此外，基于统一的训练器，CogDL 为许多有用的特性提供了原生支持，包括超参数优化、高效的训练技术和实验管理，而无需对模型实现进行任何修改。

Experiment API

CogDL 在训练上提供了更易于使用的 API ，即 Experiment 。我们以节点分类为例，展示如何使用 CogDL 完成使用 GNN 的工作流程。在监督设置中，节点分类旨在预测每个节点的真实标签。CogDL 提供了丰富的通用基准数据集和 GNN 模型。一方面，您可以使用 CogDL 中的模型和数据集简单地开始运行。当您想要测试提出的 GNN 的可复现性或在不同数据集中获得基线结果时，使用 Cogdl 很方便。

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

或者，您可以单独创建每个组件并使用 CogDL 中的 `build_dataset` , `build_model` 来手动运行该过程。

```
from cogdl import experiment
from cogdl.datasets import build_dataset
```

(continues on next page)

(continued from previous page)

```
from cogdl.models import build_model
from cogdl.options import get_default_args

args = get_default_args(model="gcn", dataset="cora")
dataset = build_dataset(args)
model = build_model(args)
experiment(model=model, dataset=dataset)
```

如上所示，模型/数据集是建立训练过程的关键组成部分。事实上，CogDL 也支持自定义模型和数据集。这将在下一章介绍。下面我们将简要介绍每个组件的详细信息。

如何保存训练好的模型？

CogDL 支持使用 `checkpoint_path` 在命令行或 API 中保存训练的模型。例如：

```
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt")
```

当训练停止时，模型将保存在 *gcn_cora.pt* 中。如果你想从之前的 checkpoint 继续训练，使用不同的参数（如学习率、权重衰减等），保持相同的模型参数（如 hidden size、模型层数），可以像下面这样做：

```
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt", resume_
↪training=True)
```

在命令行中使用 `--checkpoint-path {path}` 和 `--resume-training` 可以获得相同的结果。

如何保存 embeddings?

图表示学习（etwork embedding 和无监督 GNNs）旨在获得节点表示。embeddings 可用于各种下游应用。CogDL 会将节点 embeddings 保存在指定的路径通过 `--save-emb-path {path}`.

```
experiment(model="prone", dataset="blogcatalog", save_emb_path="./embeddings/prone_
↪blog.npy")
```

对节点分类的评估将在训练结束时进行。我们在 DeepWalk、Node2Vec 和 ProNE 中使用的相同实验设置。我们随机抽取不同百分比的标记节点来训练一个 liblinear 分类器，并将剩余的用于测试，我们重复训练几次并输出平均 Micro-F1。默认情况下，CogDL 对 90% 的标记节点进行一次抽样训练。您可以根据自己的需要使用 `--num-shuffle` 和 `--training-percents` 更改设置。

此外，CogDL 支持评估节点 embeddings，而无需在不同的评估设置中进行训练。以下代码片段评估我们在上面得到的 embeddings：

```
experiment(
    model="prone",
```

(continues on next page)

(continued from previous page)

```
    dataset="blogcatalog",
    load_emb_path="./embeddings/prone_blog.npy",
    num_shuffle=5,
    training_percents=[0.1, 0.5, 0.9]
)
```

您也可以使用命令行来实现相同的结果

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

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

2.9.6 自定义数据集

CogDL 提供了很多常见的数据集。但是您可能希望将 GNN 使用在不同应用的新数据集。CogDL 为自定义数据集提供了一个接口。你负责读取数据集，剩下的就交给 CogDL。

我们提供 `NodeDataset` and `GraphDataset` 作为抽象类并实现必要的基本操作

node_classification 的数据集

要为 node_classification 创建数据集，您需要继承 `NodeDataset`。`NodeDataset` 用于节点级预测。然后你需要实现 `process` 方法。在这种方法中，您需要读入数据并将原始数据预处理为 CogDL 可用的格式 `Graph`。之后，我们建议您保存处理后的数据（我们也会在您返回数据时帮助您保存）以避免再次进行预处理。下次运行代码时，CogDL 将直接加载它。

该模块的运行过程如下：

1. 用 *self.path* 指定保存处理数据的路径 2. 调用 *process* 函数过程来加载和预处理数据，并将您的数据保存为 *Graph* 在 *self.path* 中。此步骤将在您第一次使用数据集时实施。然后每次使用数据集时，为了方便起见，将从 *self.path* 中加载数据集。3. 对于数据集，例如节点级任务中名为 *MyNodeDataset* 的数据集，您可以通过 *MyNodeDataset.data* 或 *MyDataset[0]* 访问 data/Graph。

此外，应指定数据集评价指标。CogDL 提供 *accuracy* 和 *multiclass_f1* 用于多类别分类，*multilabel_f1* 用于多标签分类。

如果 `scale_feat` 设置为 *True* ，CogDL 将使用均值 u 和方差 s 对节点特征进行归一化：

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

这是一个 例子:

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

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

    def process(self):
        """You need to load your dataset and transform to `Graph`"""
        num_nodes, num_edges, feat_dim = 100, 300, 30

        # load or generate your dataset
        edge_index = torch.randint(0, num_nodes, (2, num_edges))
        x = torch.randn(num_nodes, feat_dim)
        y = torch.randint(0, 2, (num_nodes,))

        # set train/val/test mask in node_classification task
        train_mask = torch.zeros(num_nodes).bool()
        train_mask[0 : int(0.3 * num_nodes)] = True
        val_mask = torch.zeros(num_nodes).bool()
        val_mask[int(0.3 * num_nodes) : int(0.7 * num_nodes)] = True
        test_mask = torch.zeros(num_nodes).bool()
        test_mask[int(0.7 * num_nodes) :] = True
        data = Graph(x=x, edge_index=edge_index, y=y, train_mask=train_mask, val_
↪mask=val_mask, test_mask=test_mask)
        return data

if __name__ == "__main__":
    # Train customized dataset via defining a new class
    dataset = MyNodeDataset()
    experiment(dataset=dataset, model="gcn")

    # Train customized dataset via feeding the graph data to NodeDataset
    data = generate_random_graph(num_nodes=100, num_edges=300, num_feats=30)
    dataset = NodeDataset(data=data)
    experiment(dataset=dataset, model="gcn")
```

graph_classification 的数据集

当您要为图级别任务（例如 graph_classification ）构建数据集时，您需要继承 GraphDataset ，总体实现是相似的，而区别在于 process. 由于 GraphDataset 包含大量图，您需要将你的数据转换为 Graph 为每个图成 Graph 列表。一个例子如下所示：

```
from cogdl.data import Graph
from cogdl.datasets import GraphDataset

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))
        return graphs

if __name__ == "__main__":
    dataset = MyGraphDataset()
    experiment(model="gin", dataset=dataset)
```

2.9.7 自定义 GNN

有时您想设计自己的 GNN 模块或将 GNN 用于其他目的。在本章中，我们将介绍如何使用 CogDL 中的 GNN 层来编写自己的 GNN 模型，以及如何从头开始编写 GNN 层。

用 CogDL 中的 GNN layers 定义模型

CogDL 在 cogdl.layers 中实现了流行的 GNN 层，它们可以作为模块来帮助您设计新的 GNN。以下是我们在 CogDL 中实现 Jumping Knowledge Network (JKNet) 的 GCNLayer 方法示例。JKNet 收集所有层的输出并将它们连接在一起来获得结果：

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

```
import torch
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel

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

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

定义你的 GNN 模块

在大多数情况下，您可以使用新的消息传播和聚合方案构建层模块。这里的代码片段展示了如何在 CogDL 中使用 `Graph` 和高效的稀疏矩阵算子来实现 GCNLayer。

```
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__()
```

(continues on next page)

(continued from previous page)

```
        self.fc = torch.nn.Linear(in_feats, out_feats)

    def forward(self, graph, x):
        h = self.fc(x)
        h = spmm(graph, h)
        return h
```

spmm 是 GNN 中经常使用的稀疏矩阵乘法运算。

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

稀疏矩阵存储在 `Graph` 里，会被自动调用。空间中的消息传递等价于矩阵运算。CogDL 还支持其他高效运算符如 `edge_softmax` 和 `multi_head_spmm` 你可以参考这个 页面 使用

将自定义的 GNN 模型与 Cogdl 一起使用

现在您已经定义了自己的 GNN，您可以使用 CogDL 中的数据集/任务来立即训练和评估模型的性能。

```
data = build_dataset_from_name("cora")[0]
# Use the JKNet model as defined above
model = JKNet(data.num_features, data.num_classes, 32, 4)
experiment(model=model, dataset="cora", mw="node_classification_mw", dw="node_
↪classification_dw")
```

2.9.8 示例代码

Below is a code of examples

图简介

在 Cogdl 中表示图

```
import torch
from cogdl.data import Graph
edges = torch.tensor([[0,1],[1,3],[2,1],[4,2],[0,3]]).t()
x = torch.tensor([[-1],[0],[1],[2],[3]])
g = Graph(edge_index=edges,x=x) # equivalent to that above
print(g.row_indptr)

print(g.col_indices)
```

(continues on next page)

(continued from previous page)

```
print(g.edge_weight)

print(g.num_nodes)

print(g.num_edges)

g.edge_weight = torch.rand(5)
print(g.edge_weight)
```

如何构建 mini-batch graphs

在节点分类中，所有操作都在一个图中。但是在像图分类这样的任务中，我们需要用 mini-batch 处理很多图。图分类的数据集包含可以使用索引访问的图，例如 data [2]。为了支持小批量训练/推理，CogDL 将一批中的图组合成一个完整的图，其中邻接矩阵形成稀疏块对角矩阵，其他的（节点特征、标签）在节点维度上连接。这个过程由由 cogdl.data.Dataloader 来处理。

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

dataset = build_dataset_from_name("mutag")

print(dataset[0])

loader = DataLoader(dataset, batch_size=8)
for batch in loader:
    model(batch)
```

如何进行全局池化对每个图中节点的特征进行求和

```
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

    return out
```

如何编辑一个 graph?

在某些设置中，可以更改 edges。在这种情况下，我们需要在保留原始图的同时生成计算图。CogDL 提供了 graph.local_graph 来设置 local scape，任何 out-of-place 操作都不会反映到原始图上。但是，in-place 操作会影响原始图形。

```
graph = build_dataset_from_name("cora")[0]
print(graph.num_edges)

with graph.local_graph():
    mask = torch.arange(100)
    row, col = graph.edge_index
    graph.edge_index = (row[mask], col[mask])
    print(graph.num_edges)

print(graph.num_edges)



print(graph.edge_weight)

with graph.local_graph():
    graph.edge_weight += 1
print(graph.edge_weight)
```

Total running time of the script: (0 minutes 0.000 seconds)

模型训练

自定义模型训练逻辑

cogdl 支持选择自定义训练逻辑，“数据-模型-训练”三部分在 CogDL 中是独立的，研究者和使用者可以自定义其中任何一部分，并复用其他部分，从而提高开发效率。现在您可以使用 Cogdl 中的模型和数据集来实现您的个性化需求。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from cogdl import experiment
from cogdl.datasets import build_dataset_from_name
from cogdl.layers import GCNLayer
from cogdl.models import BaseModel
class Gnn(BaseModel):
    def __init__(self, in_feats, hidden_size, out_feats, dropout):
        super(Gnn, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.conv1 = GCNLayer(in_feats, hidden_size)
        self.conv2 = GCNLayer(hidden_size, out_feats)
        self.dropout = nn.Dropout(dropout)
    def forward(self, graph):
        graph.sym_norm()
        h = graph.x
        h = F.relu(self.conv1(graph, self.dropout(h)))
        h = self.conv2(graph, self.dropout(h))
        return F.log_softmax(h, dim=1)

if __name__ == "__main__":
    dataset = build_dataset_from_name("cora")[0]
    model = Gnn(in_feats=dataset.num_features, hidden_size=64, out_feats=dataset.num_
↪classes, dropout=0.1)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-3)
    model.train()
    for epoch in range(300):
        optimizer.zero_grad()
        out = model(dataset)
        loss = F.nll_loss(out[dataset.train_mask], dataset.y[dataset.train_mask])
        loss.backward()
        optimizer.step()
    model.eval()
    _, pred = model(dataset).max(dim=1)
    correct = float(pred[dataset.test_mask].eq(dataset.y[dataset.test_mask]).sum().
↪item())
    acc = correct / dataset.test_mask.sum().item()
    print('The accuracy rate obtained by running the experiment with the custom␣
↪training logic: {:.6f}'.format(acc))
```

Experiment API

CogDL 在训练上提供了更易于使用的 API，即 Experiment

```
from cogdl import experiment
experiment(model="gcn", dataset="cora")
# 或者，您可以单独创建每个组件并使用 CogDL 中的 build_dataset , build_model 来手动运行该过程。

from cogdl import experiment
from cogdl.datasets import build_dataset
from cogdl.models import build_model
from cogdl.options import get_default_args
```

(continues on next page)

(continued from previous page)

```
args = get_default_args(model="gcn", dataset="cora")
dataset = build_dataset(args)
model = build_model(args)
experiment(model=model, dataset=dataset)
```

如何保存训练好的模型？

```
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt")
# 当训练停止时，模型将保存在 gcn_cora.pt 中。如果你想从之前的 checkpoint 继续训练，使用不同的参数
(如学习率、权重衰减等)，保持相同的模型参数（如 hidden size、模型层数），可以像下面这样做：
experiment(model="gcn", dataset="cora", checkpoint_path="gcn_cora.pt", resume_
↪training=True)
```

如何保存 embeddings?

```
experiment(model="prone", dataset="blogcatalog", save_emb_path="./embeddings/prone_
↪blog.npy")
# 以下代码片段评估我们在上面得到的 embeddings：
experiment(
    model="prone",
    dataset="blogcatalog",
    load_emb_path="./embeddings/prone_blog.npy",
    num_shuffle=5,
    training_percents=[0.1, 0.5, 0.9]
)
```

Total running time of the script: (0 minutes 0.000 seconds)

自定义数据集

node_classification 的数据集

```
import torch
from cogdl import experiment
from cogdl.data import Graph
from cogdl.datasets import NodeDataset, generate_random_graph

class MyNodeDataset(NodeDataset):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, path="data.pt"):
        self.path = path
        super(MyNodeDataset, self).__init__(path, scale_feat=False, metric="accuracy")

    def process(self):
        """You need to load your dataset and transform to `Graph`"""
        num_nodes, num_edges, feat_dim = 100, 300, 30

        # load or generate your dataset
        edge_index = torch.randint(0, num_nodes, (2, num_edges))
        x = torch.randn(num_nodes, feat_dim)
        y = torch.randint(0, 2, (num_nodes,))

        # set train/val/test mask in node_classification task
        train_mask = torch.zeros(num_nodes).bool()
        train_mask[0 : int(0.3 * num_nodes)] = True
        val_mask = torch.zeros(num_nodes).bool()
        val_mask[int(0.3 * num_nodes) : int(0.7 * num_nodes)] = True
        test_mask = torch.zeros(num_nodes).bool()
        test_mask[int(0.7 * num_nodes) :] = True
        data = Graph(x=x, edge_index=edge_index, y=y, train_mask=train_mask, val_
↪mask=val_mask, test_mask=test_mask)
        return data

if __name__ == "__main__":
    # Train customized dataset via defining a new class
    dataset = MyNodeDataset()
    experiment(dataset=dataset, model="gcn")

    # Train customized dataset via feeding the graph data to NodeDataset
    data = generate_random_graph(num_nodes=100, num_edges=300, num_feats=30)
    dataset = NodeDataset(data=data)
    experiment(dataset=dataset, model="gcn")
```

graph_classification 的数据集

```
from cogdl.data import Graph
from cogdl.datasets import GraphDataset

class MyGraphDataset(GraphDataset):
    def __init__(self, path="data.pt"):
        self.path = path
```

(continues on next page)

(continued from previous page)

```
        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))
        return graphs

if __name__ == "__main__":
    dataset = MyGraphDataset()
    experiment(model="gin", dataset=dataset)
```

Total running time of the script: (0 minutes 0.000 seconds)

自定义 GNN

用 CogDL 中的 GNN layers 定义模型

CogDL 在 cogdl.layers 中实现了流行的 GNN 层，它们可以作为模块来帮助您设计新的 GNN。以下是我们在 CogDL 中实现 Jumping Knowledge Network (JKNet) 的 GCNLayer 方法示例。JKNet 收集所有层的输出并将它们连接在一起来获得结果：

```
import torch


from cogdl.layers import GCNLayer
from cogdl.models import BaseModel

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

    def forward(self, graph):
```

(continues on next page)

(continued from previous page)

```
        # symmetric normalization of adjacency matrix
        graph.sym_norm()
        h = graph.x
        out = []
        for layer in self.layers:
            h = layer(graph,h)
            out.append(h)
        out = torch.cat(out, dim=1)
        return self.fc(out)
```

定义你的 GNN 模块

在大多数情况下，您可以使用新的消息传播和聚合方案构建层模块。这里的代码片段展示了如何在 CogDL 中使用 Graph 和高效的稀疏矩阵算子来实现 GCNLayer。

```
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):
        h = self.fc(x)
        h = spmm(graph, h)
        return h
```

将自定义的 GNN 模型与 Cogdl 一起使用

现在您已经定义了自己的 GNN，您可以使用 CogDL 中的数据集/任务来立即训练和评估模型的性能。

```
from cogdl import experiment
from cogdl.datasets import build_dataset_from_name
data = build_dataset_from_name("cora")[0]
# Use the JKNet model as defined above
model = JKNet(data.num_features, data.num_classes, 32, 4)
experiment(model=model, dataset="cora", mw="node_classification_mw", dw="node_
↪classification_dw")
```

Total running time of the script: (0 minutes 0.000 seconds)

2.10 data

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

Bases: cogdl.data.data.BaseGraph

add_remaining_self_loops()

clone()

col_norm()

convert_csr()

degrees(*node_idx=None*)

property device

property 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()

property keys

Returns all names of graph attributes.

normalize_adj(*norm='sym'*)

property num_edges

property num_nodes

padding_self_loops()

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

remove_self_loops()

property row_indptr

row_norm()

property row_ptr_v

set_symmetric(*val*)

set_weight(*weight*)

sym_norm()

to_networkx(*weighted=True*)

to_scipy_csr()

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

Bases: *cogdl.data.data.Graph*

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

cumsum(*key*, *item*)

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

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

---

static from_data_list(*data_list*, *class_type=None*)

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

property num_graphs

Returns the number of graphs in the batch.

class cogdl.data.**DataLoader**(**args*, ***kwargs*)

Bases: `Generic`[`torch.utils.data.dataloader.T_co`]

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

Parameters

- **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)

batch_size: Optional[int]

static collate_fn(*batch*)

dataset: torch.utils.data.dataset.Dataset[torch.utils.data.dataloader.T_co]

drop_last: bool

get_parameters()

num_workers: int

pin_memory: bool

prefetch_factor: int

record_parameters(*params*)

sampler: torch.utils.data.sampler.Sampler

timeout: float

class cogdl.data.**Dataset**(*root*, *transform=None*, *pre_transform=None*, *pre_filter=None*)

Bases: Generic[torch.utils.data.dataset.T_co]

Dataset base class for creating graph datasets.

Parameters

- **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.

property edge_attr_size

get(*idx*)

Gets the data object at index `idx`.

get_evaluator()

get_loss_fn()

property max_degree

property max_graph_size

property num_classes

The number of classes in the dataset.

property num_features

Returns the number of features per node in the graph.

property num_graphs

process()

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

property processed_file_names

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

property processed_paths

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

property raw_file_names

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

property raw_paths

The filepaths to find in order to skip the download.

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

Bases: cogdl.data.data.BaseGraph

add_remaining_self_loops()

clone()

property col_indices

col_norm()

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

degrees()

property device

property edge_attr

property edge_index

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

property edge_types

property edge_weight

Return actual edge_weight

eval()

static from_dict(*dictionary*)

Creates a data object from a python dictionary.

static from_pyg_data(*data*)

property in_norm

is_inductive()

is_symmetric()

property keys

Returns all names of graph attributes.

local_graph()

mask2nid(*split*)

nodes()

normalize(*key='sym'*)

property num_classes

property num_edges

Returns the number of edges in the graph.

property num_features

Returns the number of features per node in the graph.

property num_nodes

property out_norm

padding_self_loops()

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

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

property raw_edge_weight

Return edge_weight without __in_norm__ and __out_norm__, only used for SpMM

remove_self_loops()

restore(*key*)

property row_indptr

row_norm()

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

set_asymmetric()

set_grb_adj(*adj*)

set_symmetric()

store(*key*)

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

sym_norm()

property test_nid

to_networkx()

to_scipy_csr()

train()

property train_nid

property val_nid

class cogdl.data.**MultiGraphDataset**(*root=None*, *transform=None*, *pre_transform=None*, *pre_filter=None*)

Bases: Generic[torch.utils.data.dataset.T_co]

get(*idx*)

Gets the data object at index idx.

len()

property max_degree

property max_graph_size

property num_classes

The number of classes in the dataset.

property num_features

Returns the number of features per node in the graph.

property num_graphs

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

2.11 datasets

2.11.1 GATNE dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Parameters

- **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.

property processed_file_names

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

property 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: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

2.11.2 GCC dataset

class cogdl.datasets.gcc_data.**Academic_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**DBLPNetrep_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**DBLPSnap_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

download()

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

get(*idx*)

Gets the data object at index `idx`.

property num_classes

The number of classes in the dataset.

process()

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

property processed_file_names

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

property 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.**Facebook_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**GCCDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

download()

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

get(*idx*)

Gets the data object at index `idx`.

preprocess(*root*, *name*)

property processed_file_names

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

property 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.**HIndexDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**IMDB_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**Livejournal_GCCDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.gcc_data.**PretrainDataset**(*name*, *data*)

Bases: object

class DataList(*graphs*)

Bases: object

eval()

to(*device*)

train()

get(*idx*)

get_evaluator()

get_loss_fn()

property num_features

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

2.11.3 GTN dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Parameters

- **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.

property num_classes

The number of classes in the dataset.

process()

Processes the dataset to the self.processed_dir folder.

property processed_file_names

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

property 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: Generic[torch.utils.data.dataset.T_co]

2.11.4 HAN dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.han_data.**HANDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

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

Parameters

- **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.

property num_classes

The number of classes in the dataset.

process()

Processes the dataset to the self.processed_dir folder.

property processed_file_names

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

property 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: Generic[torch.utils.data.dataset.T_co]

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

Create mask.

2.11.5 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: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.kg_data.**KnowledgeGraphDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

download()

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

get(*idx*)

Gets the data object at index `idx`.

property num_entities

property num_relations

process()

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

property processed_file_names

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

property raw_file_names

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

property test_start_idx

property train_start_idx

url = 'https://cloud.tsinghua.edu.cn/d/d1c733373b014efab986/files/?p=%2F{}%2F{}&dl=1'

property valid_start_idx

class cogdl.datasets.kg_data.**TestDataset**(*triples*, *all_true_triples*, *nentity*, *nrelation*, *mode*)

Bases: Generic[torch.utils.data.dataset.T_co]

static collate_fn(*data*)

class cogdl.datasets.kg_data.**TrainDataset**(*triples*, *nentity*, *nrelation*, *negative_sample_size*, *mode*)

Bases: Generic[torch.utils.data.dataset.T_co]

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: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

2.11.6 Matlab matrix dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Parameters

- **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`.

property num_classes

The number of classes in the dataset.

property num_nodes

process()

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

property processed_file_names

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

property 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: Generic[torch.utils.data.dataset.T_co]

download()

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

get(*idx*)

Gets the data object at index `idx`.

property num_classes

The number of classes in the dataset.

property num_nodes

process()

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

property processed_file_names

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

property 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: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

2.11.7 OGB dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBGDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

get(*idx*)

Gets the data object at index idx.

get_loader(*args*)

get_subset(*subset*)

property num_classes

The number of classes in the dataset.

class cogdl.datasets.ogb.**OGBLCitation2Dataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBLCollabDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBLDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

get(*idx*)

Gets the data object at index idx.

get_edge_split()

get_evaluator()

get_loss_fn()

property 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.**OGBLDdiDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBLPpaDataset**(*data_path='data'*)

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBNDataset**(*root*, *name*, *transform=None*)

Bases: Generic[torch.utils.data.dataset.T_co]

get(*idx*)

Gets the data object at index `idx`.

get_evaluator()

get_loss_fn()

process()

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

property 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: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.ogb.**OGBPpaDataset**

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

property edge_attr_size

get_evaluator()

get_loss_fn()

process()

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

2.11.8 TU dataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

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

Bases: Generic[torch.utils.data.dataset.T_co]

class cogdl.datasets.tu_data.**TUDataset**(*root*, *name*)

Bases: Generic[torch.utils.data.dataset.T_co]

download()

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

property num_classes

The number of classes in the dataset.

process()

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

property processed_file_names

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

property raw_file_names

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.11.9 Module contents

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

cogdl.datasets.**build_dataset_from_name**(*dataset*, *split=0*)

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

cogdl.datasets.**build_dataset_pretrain**(*args*)

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():
    (...)
```

Parameters **name** (*str*) –the name of the dataset

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

2.12 models

2.12.1 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.

property device

forward(**args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

set_loss_fn(*loss_fn*)

training: bool

2.12.2 Embedding Model

class cogdl.models.emb.hope.**HOPE**(*dimension*, *beta*)

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

- **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*)

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

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

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

The Spectral clustering model from the "Leveraging social media networks for classification" paper

Parameters **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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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.

Parameters

- **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*)

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.

---

training: bool

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.

Parameters

- **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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

the latter silently ignores them.

---

training: **bool**

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

Parameters

- **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*

forward(*graph*, *embedding_model_creator=<class 'gensim.models.word2vec.Word2Vec'>*, *return_dict=False*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

class cogdl.models.emb.gatne.**GATNE**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *epochs*, *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

Parameters

- **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.
- **epochs** (*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 "," ,
- **example** (*while each node type are connected with "-" in each metapath. For*) –"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*)

forward(*network_data*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

The Hin2vec model from the "Deep Graph Kernels" paper.

Parameters

- **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.
- **epochs** (*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*)

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

forward(*graphs*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

save_embedding(*output_path*)

training: bool

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.

Parameters

- **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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

the latter silently ignores them.

---

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

- **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.
- **epochs** (*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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_denoised_matrix(*mat*)

get_emb(*matrix*)

get_ppmi_matrix(*mat*)

random_surfing(*adj_matrix*)

scale_matrix(*mat*)

training: bool

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*)

training: bool

class cogdl.models.emb.graph2vec.**Graph2Vec**(*dimension*, *min_count*, *window_size*, *dm*, *sampling_rate*, *rounds*, *epochs*, *lr*, *worker=4*)

Bases: *cogdl.models.base_model.BaseModel*

The Graph2Vec model from the "graph2vec: Learning Distributed Representations of Graphs" paper

Parameters

- **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.
- **lr** (*float*) –Learning rate in doc2vec.

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

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

forward(*graphs*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

save_embedding(*output_path*)

training: bool

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

Parameters

- **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 "," ,
- **example** (*while each node type are connected with "-" in each metapath. For*) –"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*)

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.

---

training: bool

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

Parameters

- **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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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.

Parameters

- **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*)

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.

---

training: bool

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.

Parameters

- **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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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.

Parameters

- **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 LINE.
- **alpha** (*float*) –The initial learning rate of SGD.
- **order** (*int*) –1 represents perserving 1-st order proximity, 2 represents 2-nd,
- **them** (*while 3 means both of*) –

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

class cogdl.models.emb.sdne.**SDNE**(*hidden_size1*, *hidden_size2*, *droput*, *alpha*, *beta*, *nu1*, *nu2*, *epochs*, *lr*, *cpu*)

Bases: *cogdl.models.base_model.BaseModel*

The SDNE model from the "Structural Deep Network Embedding" paper

Parameters

- **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.
- **epochs** (*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*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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.

Parameters

- **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*)

forward(*graph:* cogdl.data.data.Graph, *return_dict=False*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

training: bool

2.12.3 GNN Model

class cogdl.models.nn.dgi.**DGIModel**(*in_feats*, *hidden_size*, *activation*)

Bases: *cogdl.models.base_model.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

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.

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

augment(*graph*)

classmethod build_model_from_args(*args*)

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

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

loss(*data*)

preprocess(*graph*)

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

The Patchy-SAN model from the "Learning Convolutional Neural Networks for Graphs" paper.

Parameters

- **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*)

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

classmethod split_dataset(*dataset*, *args*)

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

- **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*)

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.

---

training: bool

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

Parameters

- **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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data=None*)

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

reset_data(*data*)

training: bool

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*)

forward(**args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

inference(*x_all*, *data_loader*)

mini_forward(*graph*)

sampling(*edge_index*, *num_sample*)

training: bool

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

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

static add_args(*parser*)

Add model-specific arguments to the parser.

add_reverse_edges(*edge_index*, *edge_types*)

classmethod build_model_from_args(*args*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

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

predict(*graph*)

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

the latter silently ignores them.

---

predict(*graph*)

training: bool

class cogdl.models.nn.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*)

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.

---

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

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

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

training: bool

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.

Parameters

- **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*)

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

graph_classificatoin_loss(*batch*)

reset_parameters()

classmethod split_dataset(*dataset*, *args*)

training: bool

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" .

Parameters

- **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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_optimizer(*args*)

predict(*graph*)

training: bool

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*)

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

training: bool

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

Parameters

- **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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*graph*)

training: bool

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*)

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.

---

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*graph*)

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

embed(*data*)

forward(*graph: cogdl.data.data.Graph*, *x: Optional[torch.Tensor] = None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

forward(*x*, *targets*, *ppr_scores*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

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

training: bool

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?" .

Parameters

- **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*)

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

classmethod split_dataset(*dataset*, *args*)

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

- **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*)

drop_node(*x*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

normalize_x(*x*)

predict(*data*)

rand_prop(*graph*, *x*)

training: bool

class cogdl.models.nn.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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

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

normalization(*H*)

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

loss(*graph*, *scoring*)

predict(*graph*)

training: bool

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*

Implementation of DeeperGCN in paper "DeeperGCN: All You Need to Train Deeper GCNs"

Parameters

- **in_feat** (*int*) –the dimension of input features
- **hidden_size** (*int*) –the dimension of hidden representation
- **out_feat** (*int*) –the dimension of output features
- **num_layers** (*int*) –the number of layers
- **activation** (*str, optional*) –activation function. Defaults to "relu".
- **dropout** (*float, optional*) –dropout rate. Defaults to 0.0.
- **aggr** (*str, optional*) –aggregation function. Defaults to "max".
- **beta** (*float, optional*) –a coefficient for aggregation function. Defaults to 1.0.
- **p** (*float, optional*) –a coefficient for aggregation function. Defaults to 1.0.
- **learn_beta** (*bool, optional*) –whether beta is learnable. Defaults to False.
- **learn_p** (*bool, optional*) –whether p is learnable. Defaults to False.

- **learn_msg_scale** (*bool, optional*) – whether message scale is learnable. Defaults to True.
- **use_msg_norm** (*bool, optional*) – use message norm or not. Defaults to False.
- **edge_attr_size** (*int, optional*) – the dimension of edge features. Defaults to None.

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*graph*)

training: bool

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*)

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.

---

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

**Implementation 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*)

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

reset_parameters()

classmethod split_dataset(*dataset*, *args*)

sup_forward(*batch*, *x*)

training: bool

unsup_forward(*batch*, *x*)

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

Bases: *cogdl.models.base_model.BaseModel*

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

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

|------ nhidlayer ----|

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

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

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

reset_parameters()

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

the latter silently ignores them.

---

predict(*data*)

reset_parameters()

training: bool

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*)

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

training: bool

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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

training: bool

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*>__.

Parameters

- **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*)

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

classmethod split_dataset(*dataset*, *args*)

training: bool

class cogdl.models.nn.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*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*data*)

training: bool

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

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

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

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_2hop(*edge_index*)

add 2-hop neighbors as new edges

get_cluster_center()

get_features(*data*)

recon_loss(*z*, *adj*)

set_cluster_center(*center*)

training: bool

class cogdl.models.nn.agc.**AGC**(*num_clusters*, *max_iter*, *cpu*)

Bases: *cogdl.models.base_model.BaseModel*

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

Parameters

- **num_clusters** (*int*) –Number of clusters.
- **max_iter** (*int*) –Max iteration to increase k

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*args*)

compute_intra(*x*, *clusters*)

forward(*data*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

2.12.4 Model Module

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

cogdl.models.**register_model**(*name*)

New model types can be added to cogdl with the *register_model()* function decorator. For example:

```
@register_model('gat')
class GAT(BaseModel):
    (...)
```

Parameters **name** (*str*) –the name of the model

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

2.13 data wrappers

2.13.1 Node Classification

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

get_train_dataset()

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

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

get_train_dataset()

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

test_wrapper()

train_transform(*batch*)

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_transform(*batch*)

val_wrapper()

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

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

static add_args(*parser*)

get_dataset()

post_stage(*stage*, *model_w_out*)

Processing after each run

pre_stage(*stage*, *model_w_out*)

Processing before each run

pre_transform()

Data Preprocessing before all runs

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

pre_transform()

Data Preprocessing before all runs

test_wrapper()

train_wrapper() → *cogdl.data.data.Graph*

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

test_wrapper()

train_wrapper()

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

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

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

post_stage_wrapper()

pre_stage(*stage*, *model_w_out*)

Processing before each run

pre_stage_transform(*batch*)

pre_transform()

Data Preprocessing before all runs

test_transform(*batch*)

test_wrapper()

train_transform(*batch*)

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_transform(*batch*)

val_wrapper()

2.13.2 Graph Classification

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

setup_node_features()

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

pre_transform()

Data Preprocessing before all runs

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

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

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

test_wrapper()

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

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

static add_args(*parser*)

pre_transform()

Data Preprocessing before all runs

2.13.3 Pretraining

class cogdl.wrappers.data_wrapper.pretraining.**GCCDataWrapper**(*dataset*, *batch_size*, *finetune=False*, *num_workers=4*, *rw_hops=256*, *subgraph_size=128*, *restart_prob=0.8*, *positional_embedding_size=32*, *task='node_classification'*, *freeze=False*, *pretrain=False*, *num_samples=0*, *num_copies=1*, *aug='rwr'*, *num_neighbors=5*, *parallel=True*)

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

ge_wrapper()

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

2.13.4 Link Prediction

class cogdl.wrappers.data_wrapper.link_prediction.**EmbeddingLinkPredictionDataWrapper**(*dataset*, *neg-a-tive_rati*

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

static add_args(*parser*)

pre_transform()

Data Preprocessing before all runs

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

pre_transform()

Data Preprocessing before all runs

test_wrapper()

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

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

2.13.5 Heterogeneous

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

val_wrapper()

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

Bases: cogdl.wrappers.data_wrapper.base_data_wrapper.DataWrapper

test_wrapper()

train_wrapper()

Returns

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

Any other data formats other than DataLoader will not be traversed

2.14 model wrappers

2.14.1 Node Classification

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.UnsupervisedModelWrapper

static add_args(*parser*)

static augment(*graph*)

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

training: bool

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

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

static add_args(*parser*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

training: bool

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

update_soft(*graph*)

val_step(*subgraph*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.UnsupervisedModelWrapper

static add_args(*parser*)

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

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

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

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

training: bool

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

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

sample [int] Number of augmentations for consistency loss

temperature [float] Temperature to sharpen predictions.

lmbda [float] Proportion of consistency loss of unlabelled data

static add_args(*parser*)

consistency_loss(*logps*, *train_mask*)

train_step(*batch*)

training: **bool**

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.UnsupervisedModelWrapper

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

training: **bool**

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.UnsupervisedModelWrapper

static add_args(*parser*)

generate_virtual_labels(*data*)

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*graph*)

train_step(*subgraph*)

training: **bool**

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

training: bool

val_step(*batch*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.UnsupervisedModelWrapper

static add_args(*parser*)

setup_optimizer()

test_step(*graph*)

train_step(*batch*)

training: bool

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

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

static add_args(*parser*)

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

training: bool

class cogdl.wrappers.model_wrapper.node_classification.**NetworkEmbeddingModelWrapper**(*model*, *num_shuf* *train-* *ing_perce* *en-* *hance=N* *max_eval* *num_wor*

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser*)

test_step(*batch*)

train_step(*batch*)

training: bool

class cogdl.wrappers.model_wrapper.node_classification.**NodeClfModelWrapper**(*model*, *opti-* *mizer_cfg*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

set_early_stopping()

Returns

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

setup_optimizer()

test_step(*batch*)

train_step(*subgraph*)

training: bool

val_step(*subgraph*)

class cogdl.wrappers.model_wrapper.node_classification.**CorrectSmoothModelWrapper**(*model*, *op-* *ti-* *mizer_cfg*)

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

static add_args(*parser*)

test_step(*batch*)

training: bool

val_step(*subgraph*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

training: bool

val_step(*batch*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

training: bool

val_step(*batch*)

2.14.2 Graph Classification

class cogdl.wrappers.model_wrapper.graph_classification.**GraphClassificationModelWrapper**(*mod*
op-
ti-
miz

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

training: bool

val_step(*batch*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

test_step(*batch*)

train_step(*batch*)

training: bool

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

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

setup_optimizer()

sup_loss(*pred*, *batch*)

test_step(*dataset*)

train_step(*batch*)

training: bool

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

2.14.3 Pretraining

class cogdl.wrappers.model_wrapper.pretraining.**GCCModelWrapper**(*model*, *optimizer_cfg*, *nce_k*, *nce_t*, *momentum*, *output_size*, *finetune=False*, *num_classes=1*, *num_shuffle=10*, *save_model_path='saved'*, *load_model_path=''*, *freeze=False*, *pretrain=False*)

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

ge_step(*batch*)

load_checkpoint(*path*)

post_stage(*stage*, *data_w*)

pre_stage(*stage*, *data_w*)

save_checkpoint(*path*)

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

train_step_finetune(*batch*)

train_step_pretraining(*batch*)

training: **bool**

2.14.4 Link Prediction

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

test_step(*batch*)

train_step(*graph*)

training: **bool**

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

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

set_early_stopping()

Returns

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

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

training: bool

val_step(*subgraph*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

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

set_early_stopping()

Returns

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

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

training: bool

val_step(*subgraph*)

2.14.5 Heterogeneous

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser: argparse.ArgumentParser*)

Add task-specific arguments to the parser.

test_step(*batch*)

train_step(*batch*)

training: bool

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

setup_optimizer()

test_step(*batch*)

train_step(*batch*)

training: bool

val_step(*batch*)

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser: argparse.ArgumentParser*)

Add task-specific arguments to the parser.

test_step(*batch*)

train_step(*batch*)

training: bool

2.14.6 Clustering

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.EmbeddingModelWrapper

static add_args(*parser*)

test_step(*batch*)

train_step(*graph*)

training: bool

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

cluster_loss(*P*, *Q*)

getP(*Q*)

getQ(*z*, *cluster_center*)

post_stage(*stage*, *data_w*)

pre_stage(*stage*, *data_w*)

recon_loss(*z*, *adj*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

training: bool

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

Bases: cogdl.wrappers.model_wrapper.base_model_wrapper.ModelWrapper

static add_args(*parser*)

pre_stage(*stage*, *data_w*)

setup_optimizer()

test_step(*subgraph*)

train_step(*subgraph*)

training: bool

2.15 layers

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

Bases: torch.nn.modules.module.Module

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

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

reset_parameters()

training: bool

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

Bases: torch.nn.modules.module.Module

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

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

reset_parameters()

training: bool

class cogdl.layers.sage_layer.**MaxAggregator**

Bases: object

class cogdl.layers.sage_layer.**MeanAggregator**

Bases: object

class cogdl.layers.sage_layer.**SAGELayer**(*in_feats*, *out_feats*, *normalize=False*, *aggr='mean'*, *dropout=0.0*, *norm=None*, *activation=None*, *residual=False*)

Bases: torch.nn.modules.module.Module

forward(*graph*, *x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

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?" .

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

Parameters

- **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.

---

training: **bool**

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()

training: bool

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.

---

training: bool

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.

---

training: bool

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*)

training: **bool**

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"

Parameters

- **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.

---

training: **bool**

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" .

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()

training: bool

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.

---

training: bool

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.

---

training: bool

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

$$x^{(i+1)} = \sigma(W^i x^{(i)})$$

Parameters

- **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()

training: bool

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()

training: bool

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.

---

training: **bool**

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"

Parameters

- **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()

training: bool

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

training: bool

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.

training: bool

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()

training: bool

class cogdl.layers.se_layer.**SELayer**(*in_channels*, *se_channels*)

Bases: torch.nn.modules.module.Module

Squeeze-and-excitation networks

forward(*x*)

training: bool

2.16 options

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

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

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

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

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

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

cogdl.options.**get_display_data_parser**()

cogdl.options.**get_download_data_parser**()

cogdl.options.**get_parser**()

cogdl.options.**get_training_parser**()

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

2.17 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.**build_model_path**(*args*, *model_name*)

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.

Parameters

- **url** (*string*) –The url.
- **folder** (*string*) –The folder.
- **name** (*string*) –saved filename.
- **log** (*bool, optional*) –If `False`, will not print anything to the console. (default: `True`)

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

cogdl.utils.utils.**get_memory_usage**(*print_info=False*)

Get accurate gpu memory usage by querying torch runtime

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

Parameters

- **norm** –str type of normalization: *layernorm*, *batchnorm*, *instancenorm*
- **channels** –int size of features for normalization

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

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

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

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

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

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

cogdl.utils.utils.**untar**(*path*, *fname*, *deleteTar=True*)

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

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

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

Bases: `object`

clear()

evaluate()

class cogdl.utils.evaluator.**BCEWithLogitsLoss**

Bases: torch.nn.modules.module.Module

training: bool

class cogdl.utils.evaluator.**BaseEvaluator**(*eval_func*)

Bases: object

clear()

evaluate()

class cogdl.utils.evaluator.**CrossEntropyLoss**

Bases: torch.nn.modules.module.Module

training: bool

class cogdl.utils.evaluator.**MAE**

Bases: object

clear()

evaluate()

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

Bases: *cogdl.utils.evaluator.Accuracy*

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

Bases: *cogdl.utils.evaluator.Accuracy*

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

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

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

cogdl.utils.evaluator.**multiclass_f1**(*y_pred*, *y_true*)

cogdl.utils.evaluator.**multilabel_f1**(*y_pred*, *y_true*, *sigmoid=False*)

cogdl.utils.evaluator.**setup_evaluator**(*metric: Union[str, Callable]*)

class cogdl.utils.sampling.**RandomWalker**(*adj=None*, *num_nodes=None*)

Bases: object

build_up(*adj*, *num_nodes*)

walk(*start*, *walk_length*, *restart_p=0.0*, *parallel=True*)

cogdl.utils.sampling.**random_walk_parallel**(*start*, *length*, *indptr*, *indices*, *p=0.0*)

Parameters

- **start** –np.array(dtype=np.int32)
- **length** –int
- **indptr** –np.array(dtype=np.int32)
- **indices** –np.array(dtype=np.int32)
- **p** –float

Returns list(np.array(dtype=np.int32))

cogdl.utils.sampling.**random_walk_single**(*start*, *length*, *indptr*, *indices*, *p=0.0*)

Parameters

- **start** –np.array(dtype=np.int32)
- **length** –int
- **indptr** –np.array(dtype=np.int32)
- **indices** –np.array(dtype=np.int32)
- **p** –float

Returns list(np.array(dtype=np.int32))

cogdl.utils.graph_utils.**add_remaining_self_loops**(*edge_index*, *edge_weight=None*, *fill_value=1*, *num_nodes=None*)

cogdl.utils.graph_utils.**add_self_loops**(*edge_index*, *edge_weight=None*, *fill_value=1*, *num_nodes=None*)

cogdl.utils.graph_utils.**coalesce**(*row*, *col*, *value=None*)

cogdl.utils.graph_utils.**coo2csc**(*row*, *col*, *data*, *num_nodes=None*, *sorted=False*)

cogdl.utils.graph_utils.**coo2csr**(*row*, *col*, *data*, *num_nodes=None*, *ordered=False*)

cogdl.utils.graph_utils.**coo2csr_index**(*row*, *col*, *num_nodes=None*)

cogdl.utils.graph_utils.**csr2coo**(*indptr*, *indices*, *data*)

cogdl.utils.graph_utils.**csr2csc**(*indptr*, *indices*, *data=None*)

cogdl.utils.graph_utils.**get_degrees**(*row*, *col*, *num_nodes=None*)

cogdl.utils.graph_utils.**negative_edge_sampling**(*edge_index: Union[Tuple, torch.Tensor]*, *num_nodes: Optional[int] = None*, *num_neg_samples: Optional[int] = None*, *undirected: bool = False*)

cogdl.utils.graph_utils.**remove_self_loops**(*indices*, *values=None*)

cogdl.utils.graph_utils.**row_normalization**(*num_nodes*, *row*, *col*, *val=None*)

cogdl.utils.graph_utils.**sorted_coo2csr**(*row*, *col*, *data*, *num_nodes=None*, *return_index=False*)

cogdl.utils.graph_utils.**symmetric_normalization**(*num_nodes*, *row*, *col*, *val=None*)

cogdl.utils.graph_utils.**to_undirected**(*edge_index*, *num_nodes=None*)

Converts the graph given by `edge_index` to an undirected graph, so that $(j, i) \in \mathcal{E}$ for every edge $(i, j) \in \mathcal{E}$.

Parameters

- **edge_index** (*LongTensor*) –The edge indices.
- **num_nodes** (*int, optional*) –The number of nodes, *i.e.* `max_val + 1` of `edge_index`. (default: `None`)

Return type `LongTensor`

class cogdl.utils.link_prediction_utils.**ConvELayer**(*dim*, *num_filter=20*, *kernel_size=7*, *k_w=10*, *dropout=0.3*)

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

concat(*ent*, *rel*)

forward(*sub_emb*, *obj_emb*, *rel_emb*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*sub_emb*, *obj_emb*, *rel_emb*)

training: bool

class cogdl.utils.link_prediction_utils.**DistMultLayer**

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

forward(*sub_emb*, *obj_emb*, *rel_emb*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

predict(*sub_emb*, *obj_emb*, *rel_emb*)

training: bool

class cogdl.utils.link_prediction_utils.**GNNLinkPredict**

Bases: torch.nn.modules.module.Module

forward(*graph*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

get_edge_set(*edge_index*, *edge_types*)

training: bool

cogdl.utils.link_prediction_utils.**cal_mrr**(*embedding*, *rel_embedding*, *edge_index*, *edge_type*, *scoring*, *protocol='raw'*, *batch_size=1000*, *hits=[]*)

cogdl.utils.link_prediction_utils.**get_filtered_rank**(*heads*, *tails*, *rels*, *embedding*, *rel_embedding*, *batch_size*, *seen_data*)

cogdl.utils.link_prediction_utils.**get_rank**(*scores*, *target*)

cogdl.utils.link_prediction_utils.**get_raw_rank**(*heads*, *tails*, *rels*, *embedding*, *rel_embedding*, *batch_size*, *scoring*)

cogdl.utils.link_prediction_utils.**sampling_edge_uniform**(*edge_index*, *edge_types*, *edge_set*, *sampling_rate*, *num_rels*, *label_smoothing=0.0*, *num_entities=1*)

Parameters

- **edge_index** –edge index of graph
- **edge_types** –
- **edge_set** –set of all edges of the graph, (h, t, r)
- **sampling_rate** –
- **num_rels** –
- **label_smoothing** (*Optional*) –
- **num_entities** (*Optional*) –

Returns sampled existing edges rels: types of smapled existing edges sampled_edges_all: existing edges with corrupted edges sampled_types_all: types of existing and corrupted edges labels: 0/1

Return type sampled_edges

cogdl.utils.ppr_utils.**build_topk_ppr_matrix_from_data**(*edge_index*, **args*, ***kwargs*)

cogdl.utils.ppr_utils.**calc_ppr_topk_parallel**(*indptr*, *indices*, *deg*, *alpha*, *epsilon*, *nodes*, *topk*)

cogdl.utils.ppr_utils.**construct_sparse**(*neighbors*, *weights*, *shape*)

cogdl.utils.ppr_utils.**ppr_topk**(*adj_matrix*, *alpha*, *epsilon*, *nodes*, *topk*)

Calculate the PPR matrix approximately using Anderson.

cogdl.utils.ppr_utils.**topk_ppr_matrix**(*adj_matrix*, *alpha*, *eps*, *idx*, *topk*, *normalization='row'*)

Create a sparse matrix where each node has up to the topk PPR neighbors and their weights.

class cogdl.utils.prone_utils.**Gaussian**(*mu=0.5*, *theta=1*, *rescale=False*, *k=3*)

Bases: object

prop(*mx*, *emb*)

class cogdl.utils.prone_utils.**HeatKernel**(*t=0.5*, *theta0=0.6*, *theta1=0.4*)

Bases: object

prop(*mx*, *emb*)

prop_adjacency(*mx*)

class cogdl.utils.prone_utils.**HeatKernelApproximation**(*t=0.2*, *k=5*)

Bases: object

chebyshev(*mx*, *emb*)

prop(*mx*, *emb*)

taylor(*mx*, *emb*)

class cogdl.utils.prone_utils.**NodeAdaptiveEncoder**

Bases: object

- shrink negative values in signal/feature matrix
- no learning

static prop(*signal*)

class cogdl.utils.prone_utils.**PPR**(*alpha=0.5*, *k=10*)

Bases: object

applying sparsification to accelerate computation

prop(*mx*, *emb*)

class cogdl.utils.prone_utils.**ProNE**

Bases: object

class cogdl.utils.prone_utils.**SignalRescaling**

Bases: object

- **rescale signal of each node according to the degree of the node:**
  - sigmoid(degree)
  - sigmoid(1/degree)

prop(*mx*, *emb*)

cogdl.utils.prone_utils.**get_embedding_dense**(*matrix*, *dimension*)

cogdl.utils.prone_utils.**propagate**(*mx*, *emb*, *stype*, *space=None*)

class cogdl.utils.srgcn_utils.**ColumnUniform**

Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

class cogdl.utils.srgcn_utils.**EdgeAttention**(*in_feat*)

Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: bool

class cogdl.utils.srgcn_utils.**Gaussian**(*in_feat*)

Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

class cogdl.utils.srgcn_utils.**HeatKernel**(*in_feat*)

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

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

class cogdl.utils.srgcn_utils.**Identity**(*in_feat*)

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

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

class cogdl.utils.srgcn_utils.**NodeAttention**(*in_feat*)

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

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

training: **bool**

class cogdl.utils.srgcn_utils.**NormIdentity**

Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

training: **bool**

class cogdl.utils.srgcn_utils.**PPR**(*in_feat*)

Bases: torch.nn.modules.module.Module

forward(*x*, *edge_index*, *edge_attr*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

training: **bool**

class cogdl.utils.srgcn_utils.**RowSoftmax**

Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

the latter silently ignores them.

---

training: **bool**

class cogdl.utils.srgcn_utils.**RowUniform**

Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

class cogdl.utils.srgcn_utils.**SymmetryNorm**

Bases: torch.nn.modules.module.Module

forward(*edge_index*, *edge_attr*, *N*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

training: **bool**

cogdl.utils.srgcn_utils.**act_attention**(*attn_type*)

cogdl.utils.srgcn_utils.**act_map**(*act*)

cogdl.utils.srgcn_utils.**act_normalization**(*norm_type*)

2.18 experiments

class cogdl.experiments.**AutoML**(*args*)

Bases: object

Parameters **search_space** –function to obtain hyper-parameters to search

run()

cogdl.experiments.**auto_experiment**(*args*)

cogdl.experiments.**default_search_space**(*trial*)

cogdl.experiments.**experiment**(*dataset*, *model=None*, ***kwargs*)

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

cogdl.experiments.**getpid**(*_*)

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

cogdl.experiments.**raw_experiment**(*args*)

cogdl.experiments.**set_best_config**(*args*)

cogdl.experiments.**train**(*args*)

cogdl.experiments.**train_parallel**(*args*)

cogdl.experiments.**variant_args_generator**(*args*, *variants*)

Form variants as group with size of num_workers

2.19 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

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y