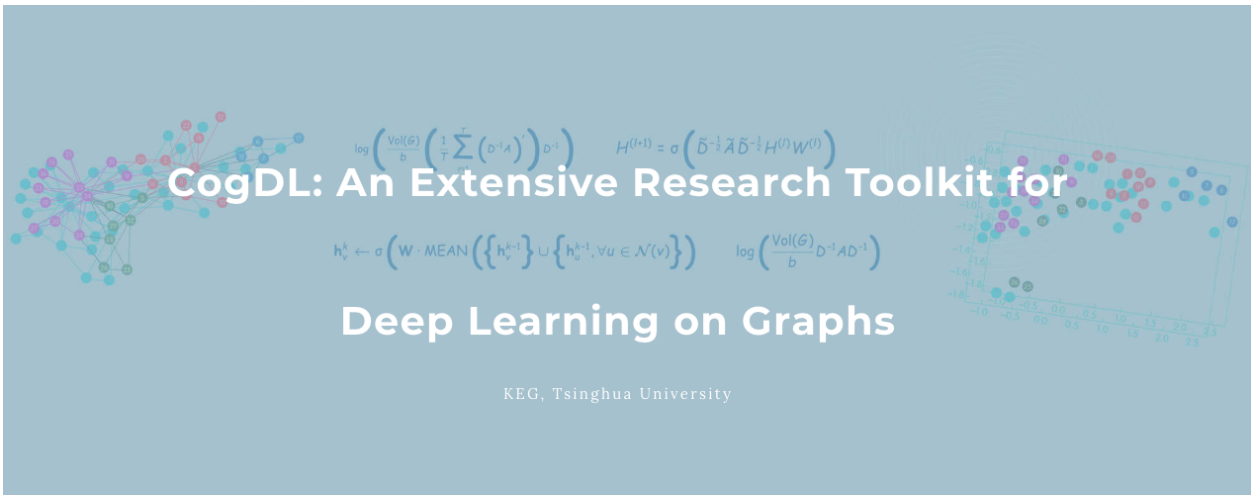
CogDL Documentation

KEG

Mar 03, 2021

CONTENTS:



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

CogDL provides these features:

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

CHAPTER

ONE

INSTALL

- PyTorch version >= 1.0.0
- Python version >= 3.6
- PyTorch Geometric (optional)

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

Please follow the instructions here to install PyTorch Geometric: https://github.com/rusty1s/pytorch_geometric/#installation.

Install other dependencies:

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

CHAPTER

TWO

TUTORIAL

This guide can help you start working with CogDL.

2.1 Create a model

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

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

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

Then we use function decorator to declare new model for CogDL

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

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

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

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

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

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

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

2.2 Create a dataset

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

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

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

2.3 Create a task

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

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

    def __init__(self, args):
        pass

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

2.4 Combine model, dataset and task

After create your model, dataset and task, we could combine them together to learn representation from a model on a dataset and evaluate its performance according to a task. We use 'build_model', 'build_dataset', 'build_task' method to build them with cooresponding parameters.

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

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

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

(continues on next page)

(continued from previous page)

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

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

CHAPTER

THREE

TASKS

3.1 Node Classification

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

First we define the *NodeClassification* class.

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

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

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

Then we can build dataset according to args.

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

To run NodeClassification, we can use the following command:

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

Then We get experimental results like this:

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

3.2 Unsupervised Node Classification

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

self.save_emb(features_matrix)

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        for train_percent in training_percents:
            for shuf in shuffles:
```

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

```
X, y = shuf

training_size = int(train_percent * self.num_nodes)

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

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

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

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

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

```
from sklearn.multiclass import OneVsRestClassifier

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

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

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

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

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

To run UnsupervisedNodeClassification, we can use following instruction:

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

Then We get experimental results like this:

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

3.3 Supervised Graph Classification

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

Task Design

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

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

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

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

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

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

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

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

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

Create a model

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

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

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

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

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

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

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

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

Run

To run GraphClassification, we can use the following command:

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

Then We get experimental results like this:

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

3.4 Unsupervised Graph Classification

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

Task Design

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
            # ...
```

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

Create a model

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

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

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

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

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

Run

To run UnsupervisedGraphClassification, we can use the following command:

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

Then we get experimental results like this:

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

CHAPTER

FOUR

LICENSE

MIT License

Copyright (c) 2020

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

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

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

CHAPTER

FIVE

CITING

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

CHAPTER

SIX

API REFERENCE

This page contains auto-generated API reference documentation[1].

6.1 options

6.1.1 Module Contents

Functions

get_parser()	
add_task_args(parser)	
add_dataset_args(parser)	
add_model_args(parser)	
get_training_parser()	
get_display_data_parser()	
get_download_data_parser()	
parse_args_and_arch(parser, args)	The parser doesn't know about model-specific args, so we parse twice.

options.**get_parser**()

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

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

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

options.**get_training_parser**()

options.**get_display_data_parser**()

options.**get_download_data_parser**()

[1] Created with sphinx-autoapi

options.**parse_args_and_arch**(*parser*, *args*)
The parser doesn't know about model-specific args, so we parse twice.

6.2 utils

6.2.1 Module Contents

Classes

ArgClass	

Functions

build_args_from_dict(dic)	
add_remaining_self_loops(edge_index, edge_weight, fill_value, num_nodes)	

class utils.**ArgClass**
Bases: object

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

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

utils.**args**

6.3 layers

6.3.1 Submodules

layers.gcc_module

Module Contents

Classes

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

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

Squeeze-and-excitation networks

forward(*self*, *x*)

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

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

forward(*self*, *h*)

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

MLP with linear output

forward(*self*, *x*)

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

forward(*self*, *g*, *n_feat*, *e_feat*)

class layers.gcc_module.**UnsupervisedMPNN**(*output_dim=32*, *node_input_dim=32*, *node_hidden_dim=32*, *edge_input_dim=32*, *edge_hidden_dim=32*, *num_step_message_passing=6*, *lstm_as_gate=False*)
Bases: torch.nn.Module

MPNN from Neural Message Passing for Quantum Chemistry

node_input_dim [int] Dimension of input node feature, default to be 15.

edge_input_dim [int] Dimension of input edge feature, default to be 15.

output_dim [int] Dimension of prediction, default to be 12.

node_hidden_dim [int] Dimension of node feature in hidden layers, default to be 64.

edge_hidden_dim [int] Dimension of edge feature in hidden layers, default to be 128.

num_step_message_passing [int] Number of message passing steps, default to be 6.

num_step_set2set [int] Number of set2set steps

num_layer_set2set [int] Number of set2set layers

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

g [DGLGraph] Input DGLGraph for molecule(s)

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

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

res : Predicted labels

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

Bases: torch.nn.Module

GIN model

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

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

Bases: torch.nn.Module

MPNN from Neural Message Passing for Quantum Chemistry

node_input_dim [int] Dimension of input node feature, default to be 15.

edge_input_dim [int] Dimension of input edge feature, default to be 15.

output_dim [int] Dimension of prediction, default to be 12.

node_hidden_dim [int] Dimension of node feature in hidden layers, default to be 64.

edge_hidden_dim [int] Dimension of edge feature in hidden layers, default to be 128.

num_step_message_passing [int] Number of message passing steps, default to be 6.

num_step_set2set [int] Number of set2set steps

num_layer_set2set [int] Number of set2set layers

forward(*self*, *g*, *return_all_outputs=False*)

Predict molecule labels

g [DGLGraph] Input DGLGraph for molecule(s)

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

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

res : Predicted labels

layers.maggregator

Module Contents

Classes

MeanAggregator

class layers.maggregator.**MeanAggregator**(*in_channels*, *out_channels*, *improved=False*, *cached=False*, *bias=True*)

Bases: torch.nn.Module

static norm(*x*, *edge_index*)

forward(*self*, *x*, *edge_index*, *edge_weight=None*, *bias=True*)

update(*self*, *aggr_out*)

__repr__(*self*)

layers.mixhop_layer

Module Contents

Classes

MixHopLayer	

class layers.mixhop_layer.**MixHopLayer**(*num_features*, *adj_pows*, *dim_per_pow*)

Bases: torch.nn.Module

reset_parameters(*self*)

adj_pow_x(*self*, *x*, *adj*, *p*)

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

layers.mixhop_layer.**layer**

layers.se_layer

Module Contents

Classes

SELayer	Squeeze-and-excitation networks

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

Bases: torch.nn.Module

Squeeze-and-excitation networks

forward(*self*, *x*)

layers.srgcn_module

Module Contents

Classes

NodeAttention
EdgeAttention
Identity
Gaussian
PPR
HeatKernel
NormIdentity
RowUniform
RowSoftmax
ColumnUniform
SymmetryNorm

Functions

act_attention(attn_type)
act_normalization(norm_type)
act_map(act)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

forward(*self*, *x*, *edge_index*, *edge_attr*)

layers.srgcn_module.**act_attention**(*attn_type*)

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

forward(*self*, *edge_index*, *edge_attr*, *N*)

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

forward(*self*, *edge_index*, *edge_attr*, *N*)

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

forward(*self*, *edge_index*, *edge_attr*, *N*)

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

forward(*self*, *edge_index*, *edge_attr*, *N*)

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

forward(*self*, *edge_index*, *edge_attr*, *N*)

layers.srgcn_module.**act_normalization**(*norm_type*)

layers.srgcn_module.**act_map**(*act*)

6.3.2 Package Contents

Classes

MeanAggregator	
SELayer	Squeeze-and-excitation networks
MixHopLayer	

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

static norm(*x*, *edge_index*)

forward(*self*, *x*, *edge_index*, *edge_weight=None*, *bias=True*)

update(*self*, *aggr_out*)

__repr__(*self*)

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

Squeeze-and-excitation networks

forward(*self*, *x*)

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

reset_parameters(*self*)

adj_pow_x(*self*, *x*, *adj*, *p*)

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

6.4 data

6.4.1 Submodules

data.batch

Module Contents

Classes

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

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

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

static from_data_list(*data_list*, *follow_batch=[]*)
Constructs a batch object from a python list holding torch_geometric.data.Data objects. The assignment vector batch is created on the fly. Additionally, creates assignment batch vectors for each key in follow_batch.

cumsum(*self*, *key*, *item*)
If True, the attribute key with content item should be added up cumulatively before concatenated together.

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

to_data_list(*self*)

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

property num_graphs(*self*)
Returns the number of graphs in the batch.

data.data

Module Contents

Classes

Data	A plain old python object modeling a single graph with various

class data.data.**Data**(*x=None*, *edge_index=None*, *edge_attr=None*, *y=None*, *pos=None*)
Bases: `object`

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

Args:

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

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

edge_attr (Tensor, optional): Edge feature matrix with shape `[num_edges, num_edge_features]`. (default: `None`)

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

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

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

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

__getitem__(*self*, *key*)
Gets the data of the attribute `key`.

__setitem__(*self*, *key*, *value*)
Sets the attribute `key` to `value`.

property keys(*self*)
Returns all names of graph attributes.

__len__(*self*)
Returns the number of all present attributes.

__contains__(*self*, *key*)
Returns `True`, if the attribute `key` is present in the data.

__iter__(*self*)
Iterates over all present attributes in the data, yielding their attribute names and content.

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

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

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

__inc__(*self*, *key*, *value*)
"Returns the incremental count to cumulatively increase the value of the next attribute of `key` when creating batches.

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

---

property num_edges(*self*)
Returns the number of edges in the graph.

property num_features(*self*)
Returns the number of features per node in the graph.

property num_nodes(*self*)

is_coalesced(*self*)
Returns `True`, if edge indices are ordered and do not contain duplicate entries.

apply(*self*, *func*, **keys*)
Applies the function `func` to all attributes `*keys`. If `*keys` is not given, `func` is applied to all present attributes.

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

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

cuda(*self*, **keys*)

clone(*self*)

__repr__(*self*)
Return repr(self).

data.dataloader

Module Contents

Classes

DataLoader	Data loader which merges data objects from a
DataListLoader	Data loader which merges data objects from a
DenseDataLoader	Data loader which merges data objects from a

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

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

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

(default: 1)

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

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

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

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

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

(default: 1)

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

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

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

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

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

(default: 1)

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

data.dataset

Module Contents

Classes

Dataset	Dataset base class for creating graph datasets.

Functions

to_list(x)	
files_exist(files)	

data.dataset.**to_list**(*x*)

data.dataset.**files_exist**(*files*)

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

Bases: torch.utils.data.Dataset

Dataset base class for creating graph datasets. See here for the accompanying tutorial.

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

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

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

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

property raw_file_names(*self*)

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

property processed_file_names(*self*)

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

abstract download(*self*)

Downloads the dataset to the self.raw_dir folder.

abstract process(*self*)

Processes the dataset to the self.processed_dir folder.

abstract __len__(*self*)

The number of examples in the dataset.

abstract get(*self*, *idx*)

Gets the data object at index idx.

property num_features(*self*)
Returns the number of features per node in the graph.

property raw_paths(*self*)
The filepaths to find in order to skip the download.

property processed_paths(*self*)
The filepaths to find in the `self.processed_dir` folder in order to skip the processing.

_download(*self*)

_process(*self*)

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

__repr__(*self*)

data.download

Module Contents

Functions

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

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

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

console. (default: `True`)

data.extract

Module Contents

Functions

maybe_log(path, log=True)	
extract_tar(path, folder, mode='r:gz', log=True)	Extracts a tar archive to a specific folder.
extract_zip(path, folder, log=True)	Extracts a zip archive to a specific folder.
extract_bz2(path, folder, log=True)	
extract_gz(path, folder, log=True)	

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

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

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

console. (default: `True`)

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

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

console. (default: `True`)

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

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

data.makedirs

Module Contents

Functions

makedirs(path)	

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

6.4.2 Package Contents

Classes

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

Functions

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

class data.**Data**(*x=None*, *edge_index=None*, *edge_attr=None*, *y=None*, *pos=None*)
Bases: object

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

Args:

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

edge_index (LongTensor, optional): Graph connectivity in COO format with shape [2, num_edges]. (default: None)

edge_attr (Tensor, optional): Edge feature matrix with shape [num_edges, num_edge_features]. (default: None)

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

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

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

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

__getitem__(*self*, *key*)
Gets the data of the attribute key.

__setitem__(*self*, *key*, *value*)
Sets the attribute key to value.

property keys(*self*)
Returns all names of graph attributes.

__len__(*self*)
Returns the number of all present attributes.

__contains__(*self*, *key*)
Returns True, if the attribute key is present in the data.

__iter__(*self*)
Iterates over all present attributes in the data, yielding their attribute names and content.

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

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

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

__inc__(*self*, *key*, *value*)
"Returns the incremental count to cumulatively increase the value of the next attribute of key when creating batches.

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

property num_edges(*self*)
Returns the number of edges in the graph.

property num_features(*self*)
Returns the number of features per node in the graph.

property num_nodes(*self*)

is_coalesced(*self*)
Returns `True`, if edge indices are ordered and do not contain duplicate entries.

apply(*self*, *func*, **keys*)
Applies the function `func` to all attributes `*keys`. If `*keys` is not given, `func` is applied to all present attributes.

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

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

cuda(*self*, **keys*)

clone(*self*)

__repr__(*self*)
Return repr(self).

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

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

static from_data_list(*data_list*, *follow_batch=[]*)
Constructs a batch object from a python list holding `torch_geometric.data.Data` objects. The assignment vector *batch* is created on the fly. Additionally, creates assignment batch vectors for each key in `follow_batch`.

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

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

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

property num_graphs(*self*)
Returns the number of graphs in the batch.

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

Dataset base class for creating graph datasets. See here for the accompanying tutorial.

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

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

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

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

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

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

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

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

abstract __len__(*self*)
The number of examples in the dataset.

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

property num_features(*self*)
Returns the number of features per node in the graph.

property raw_paths(*self*)
The filepaths to find in order to skip the download.

property processed_paths(*self*)
The filepaths to find in the self.processed_dir folder in order to skip the processing.

_download(*self*)

_process(*self*)

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

__repr__(*self*)

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

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

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

(default: 1)

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

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

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

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

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

(default: `1`)

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

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

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

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

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

(default: `1`)

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

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

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

console. (default: `True`)

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

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

console. (default: `True`)

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

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

console. (default: `True`)

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

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

6.5 tasks

6.5.1 Submodules

tasks.base_task

Module Contents

Classes

BaseTask	

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

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

abstract train(*self*, *num_epoch*)

tasks.graph_classification

Module Contents

Classes

GraphClassification	Superiviced graph classification task.

Functions

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

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

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

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

Superiviced graph classification task.

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

train(*self*)

_train(*self*)

_train_step(*self*)

_test_step(*self*, *split='val'*)

_kfold_train(*self*)

generate_data(*self*, *dataset*, *args*)

tasks.heterogeneous_node_classification

Module Contents

Classes

HeterogeneousNodeClassification	Heterogeneous Node classification task.

class tasks.heterogeneous_node_classification.**HeterogeneousNodeClassification**(*args*)

Bases: *tasks.BaseTask*

Heterogeneous Node classification task.

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

_train_step(*self*)

_test_step(*self*, *split='val'*)

tasks.link_prediction

Module Contents

Classes

LinkPrediction	

Functions

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

continues on next page

Table 26 – continued from previous page

evaluate(embs, true_edges, false_edges)	

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

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

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

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

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

class tasks.link_prediction.**LinkPrediction**(*args*)

Bases: *tasks.BaseTask*

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

tasks.multiplex_link_prediction

Module Contents

Classes

MultiplexLinkPrediction	

Functions

get_score(embs, node1, node2)	
evaluate(embs, true_edges, false_edges)	

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

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

class tasks.multiplex_link_prediction.**MultiplexLinkPrediction**(*args*)

Bases: *tasks.BaseTask*

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

tasks.multiplex_node_classification

Module Contents

Classes

MultiplexNodeClassification	Node classification task.

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

Bases: *tasks.BaseTask*

Node classification task.

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

tasks.node_classification

Module Contents

Classes

NodeClassification	Node classification task.

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

Bases: *tasks.BaseTask*

Node classification task.

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

_train_step(*self*)

_test_step(*self*, *split='val'*)

tasks.node_classification_sampling

Module Contents

Classes

NodeClassificationSampling	Node classification task with sampling.

Functions

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

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

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

Bases: *tasks.BaseTask*

Node classification task with sampling.

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

_train_step(*self*)

_test_step(*self*, *split='val'*)

tasks.unsupervised_graph_classification

Module Contents

Classes

UnsupervisedGraphClassification	Unsupervised graph classification

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

Bases: *tasks.BaseTask*

Unsupervised graph classification

static add_args(*parser*)

Add task-specific arguments to the parser.

train(*self*)

save_emb(*self*, *embs*)

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

tasks.unsupervised_node_classification

Module Contents

Classes

UnsupervisedNodeClassification	Node classification task.
TopKRanker	

tasks.unsupervised_node_classification.**pyg = False**

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

Node classification task.

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

enhance_emb(*self*, *G*, *embs*)

save_emb(*self*, *embs*)

train(*self*)

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

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

predict(*self*, *X*, *top_k_list*)

6.5.2 Package Contents

Classes

BaseTask	

Functions

register_task(name)	New task types can be added to cogdl with the *register_task()*
build_task(args, dataset=None, model=None)	

class tasks.**BaseTask**(*args*)
Bases: object

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

abstract train(*self*, *num_epoch*)

tasks.**TASK_REGISTRY**

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

For example:

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

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

tasks.**task_name**

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

6.6 datasets

6.6.1 Submodules

datasets.gatne

Module Contents

Classes

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

Functions

read_gatne_data(folder)	

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

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

Bases: cogdl.data.Dataset

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

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

"Twitter", "YouTube").

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

property raw_file_names(*self*)

property processed_file_names(*self*)

get(*self*, *idx*)

download(*self*)

process(*self*)

__repr__(*self*)

class datasets.gatne.**AmazonDataset**
Bases: *datasets.gatne.GatneDataset*

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

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

`"Twitter"`, `"YouTube"`).

class datasets.gatne.**TwitterDataset**
Bases: *datasets.gatne.GatneDataset*

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

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

`"Twitter"`, `"YouTube"`).

class datasets.gatne.**YouTubeDataset**
Bases: *datasets.gatne.GatneDataset*

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

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

`"Twitter"`, `"YouTube"`).

datasets.gcc_data

Module Contents

Classes

Edgelist
USAAirportDataset

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

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

property raw_file_names(*self*)

property processed_file_names(*self*)

download(*self*)

get(*self*, *idx*)

process(*self*)

class datasets.gcc_data.**USAAirportDataset**

Bases: *datasets.gcc_data.Edgelist*

datasets.gtn_data

Module Contents

Classes

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

Functions

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

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

class datasets.gtn_data.**GTNDataset**(*root*, *name*)
Bases: cogdl.data.Dataset

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

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

`"gtn-dblp"`, `"gtn-imdb"`).

property raw_file_names(*self*)

property processed_file_names(*self*)

read_gtn_data(*self*, *folder*)

get(*self*, *idx*)

apply_to_device(*self*, *device*)

download(*self*)

process(*self*)

__repr__(*self*)

class datasets.gtn_data.**ACM_GTNDataset**
Bases: *datasets.gtn_data.GTNDataset*

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

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

"gtn-dblp", "gtn-imdb").

class datasets.gtn_data.**DBLP_GTNDataset**
Bases: *datasets.gtn_data.GTNDataset*

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

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

"gtn-dblp", "gtn-imdb").

class datasets.gtn_data.**IMDB_GTNDataset**
Bases: *datasets.gtn_data.GTNDataset*

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

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

"gtn-dblp", "gtn-imdb").

datasets.han_data

Module Contents

Classes

HANDataset	The network datasets "ACM", "DBLP" and "IMDB" from the
ACM_HANDataset	The network datasets "ACM", "DBLP" and "IMDB" from the
DBLP_HANDataset	The network datasets "ACM", "DBLP" and "IMDB" from the
IMDB_HANDataset	The network datasets "ACM", "DBLP" and "IMDB" from the

Functions

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

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

datasets.han_data.**sample_mask**(*idx*, *l*)
Create mask.

class datasets.han_data.**HANDataset**(*root*, *name*)
Bases: cogdl.data.Dataset

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

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

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

property raw_file_names(*self*)

property processed_file_names(*self*)

read_gtn_data(*self*, *folder*)

get(*self*, *idx*)

apply_to_device(*self*, *device*)

download(*self*)

process(*self*)

__repr__(*self*)

class datasets.han_data.**ACM_HANDataset**
Bases: *datasets.han_data.HANDataset*

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

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

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

class datasets.han_data.**DBLP_HANDataset**
Bases: *datasets.han_data.HANDataset*

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

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

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

class datasets.han_data.**IMDB_HANDataset**
Bases: *datasets.han_data.HANDataset*

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

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

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

datasets.matlab_matrix

Module Contents

Classes

MatlabMatrix	networks from the http://leitang.net/code/social-dimension/data/ or http://snap.stanford.edu/node2vec/
BlogcatalogDataset	networks from the http://leitang.net/code/social-dimension/data/ or http://snap.stanford.edu/node2vec/
FlickrDataset	networks from the http://leitang.net/code/social-dimension/data/ or http://snap.stanford.edu/node2vec/
WikipediaDataset	networks from the http://leitang.net/code/social-dimension/data/ or http://snap.stanford.edu/node2vec/
PPIDataset	networks from the http://leitang.net/code/social-dimension/data/ or http://snap.stanford.edu/node2vec/

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

Bases: cogdl.data.Dataset

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

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

property raw_file_names(*self*)

property processed_file_names(*self*)

download(*self*)

get(*self*, *idx*)

process(*self*)

class datasets.matlab_matrix.**BlogcatalogDataset**

Bases: *datasets.matlab_matrix.MatlabMatrix*

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

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

class datasets.matlab_matrix.**FlickrDataset**

Bases: *datasets.matlab_matrix.MatlabMatrix*

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

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

class datasets.matlab_matrix.**WikipediaDataset**

Bases: *datasets.matlab_matrix.MatlabMatrix*

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

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

class datasets.matlab_matrix.**PPIDataset**

Bases: *datasets.matlab_matrix.MatlabMatrix*

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

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

datasets.pyg

Module Contents

Classes

CoraDataset
CiteSeerDataset
PubMedDataset
RedditDataset
MUTAGDataset
ImdbBinaryDataset
ImdbMultiDataset
CollabDataset
ProtainsDataset
RedditBinary
RedditMulti5K
RedditMulti12K
PTCMRDataset
NCT1Dataset
NCT109Dataset
ENZYMES
QM9Dataset

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

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

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

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

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

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

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

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

class datasets.pyg.**ProtainsDataset**
Bases: torch_geometric.datasets.TUDataset

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

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

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

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

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

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

class datasets.pyg.**ENZYMES**
Bases: torch_geometric.datasets.TUDataset

__getitem__(*self*, *idx*)

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

datasets.pyg_modelnet

Module Contents

Classes

ModelNet10
ModelNet40
ModelNetData10
ModelNetData40

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

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

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

get_all(*self*)

__getitem__(*self*, *item*)

__len__(*self*)

property train_index(*self*)

property test_index(*self*)

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

get_all(*self*)

__getitem__(*self*, *item*)

__len__(*self*)

property train_index(*self*)

property test_index(*self*)

6.6.2 Package Contents

Functions

register_dataset(name)	New dataset types can be added to cogdl with the *register_dataset()*
build_dataset(args)	
build_dataset_from_name(dataset)	

datasets.**pyg = False**

datasets.**dgl_import = False**

datasets.**DATASET_REGISTRY**

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

For example:

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

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

datasets.**dataset_name**

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

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

6.7 models

6.7.1 Subpackages

models.emb

Submodules

models.emb.deepwalk

Module Contents

Classes

DeepWalk	The DeepWalk model from the `"DeepWalk: Online Learning of Social Representations"

class models.emb.deepwalk.**DeepWalk**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *iteration*)

Bases: *models.BaseModel*

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

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

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

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

train(*self*, *G*)

_walk(*self*, *start_node*, *walk_length*)

_simulate_walks(*self*, *walk_length*, *num_walks*)

models.emb.dgk

Module Contents

Classes

DeepGraphKernel	The Hin2vec model from the `"Deep Graph Kernels"

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

Bases: *models.BaseModel*

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

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

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

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

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

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

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

save_embedding(*self*, *output_path*)

models.emb.dngr

Module Contents

Classes

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

class models.emb.dngr.**DNGR_layer**(*num_node*, *hidden_size1*, *hidden_size2*)

Bases: torch.nn.Module

forward(*self*, *x*)

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

Bases: *models.BaseModel*

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

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

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

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

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

scale_matrix(*self*, *mat*)

random_surfing(*self*, *adj_matrix*)

get_ppmi_matrix(*self*, *mat*)

get_denoised_matrix(*self*, *mat*)

get_emb(*self*, *matrix*)

train(*self*, *G*)

models.emb.gatne

Module Contents

Classes

GATNE	The GATNE model from the `"Representation Learning for Attributed Multiplex Heterogeneous Network"
GATNEModel	
NSLoss	
RWGraph	

Functions

get_G_from_edges(edges)	
generate_pairs(all_walks, vocab, window_size=5)	
generate_vocab(all_walks)	
get_batches(pairs, neighbors, batch_size)	
generate_walks(network_data, num_walks, walk_length, schema=None)	

class models.emb.gatne.**GATNE**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *epoch*, *batch_size*, *edge_dim*, *att_dim*, *negative_samples*, *neighbor_samples*, *schema*)

Bases: *models.BaseModel*

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

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

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

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

train(*self*, *network_data*)

class models.emb.gatne.**GATNEModel**(*num_nodes*, *embedding_size*, *embedding_u_size*, *edge_type_count*, *dim_a*)
Bases: torch.nn.Module

reset_parameters(*self*)

forward(*self*, *train_inputs*, *train_types*, *node_neigh*)

class models.emb.gatne.**NSLoss**(*num_nodes*, *num_sampled*, *embedding_size*)
Bases: torch.nn.Module

reset_parameters(*self*)

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

class models.emb.gatne.**RWGraph**(*nx_G*, *node_type=None*)

walk(*self*, *walk_length*, *start*, *schema=None*)

simulate_walks(*self*, *num_walks*, *walk_length*, *schema=None*)

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

models.emb.gatne.**generate_pairs**(*all_walks*, *vocab*, *window_size=5*)

models.emb.gatne.**generate_vocab**(*all_walks*)

models.emb.gatne.**get_batches**(*pairs*, *neighbors*, *batch_size*)

models.emb.gatne.**generate_walks**(*network_data*, *num_walks*, *walk_length*, *schema=None*)

models.emb.graph2vec

Module Contents

Classes

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

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

Bases: *models.BaseModel*

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

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

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

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

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

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

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

save_embedding(*self*, *output_path*)

models.emb.grarep

Module Contents

Classes

GraRep	The GraRep model from the ``"Grarep: Learning graph representations with global structural information"

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

Bases: *models.BaseModel*

The GraRep model from the "Grarep: Learning graph representations with global structural information" paper.

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

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

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

train(*self*, *G*)

_get_embedding(*self*, *matrix*, *dimension*)

models.emb.hin2vec

Module Contents

Classes

Hin2vec_layer	
RWgraph	
Hin2vec	The Hin2vec model from the `"HIN2Vec: Explore Meta-paths in Heterogeneous Information Networks for Representation Learning"

class models.emb.hin2vec.**Hin2vec_layer**(*num_node*, *num_relation*, *hidden_size*, *cpu*)

Bases: torch.nn.Module

regulartion(*self*, *embr*)

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

get_emb(*self*)

class models.emb.hin2vec.**RWgraph**(*nx_G*, *node_type=None*)

_walk(*self*, *start_node*, *walk_length*)

_simulate_walks(*self*, *walk_length*, *num_walks*)

data_preparation(*self*, *walks*, *hop*, *negative*)

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

Bases: *models.BaseModel*

The Hin2vec model from the "HIN2Vec: Explore Meta-paths in Heterogeneous Information Networks for Representation Learning" paper.

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

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

models.emb.hope

Module Contents

Classes

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

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

Bases: *models.BaseModel*

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

Args: hidden_size (int) : The dimension of node representation. beta (float) : Parameter in katz decomposition.

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

train(*self*, *G*)

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

_get_embedding(*self*, *matrix*, *dimension*)

models.emb.line

Module Contents

Classes

LINE	The LINE model from the `"Line: Large-scale information network embedding"

class models.emb.line.**LINE**(*dimension*, *walk_length*, *walk_num*, *negative*, *batch_size*, *alpha*, *order*)

Bases: *models.BaseModel*

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

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

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

train(*self*, *G*)

_update(*self*, *vec_u*, *vec_v*, *vec_error*, *label*)

_train_line(*self*, *order*)

models.emb.metapath2vec

Module Contents

Classes

Metapath2vec	The Metapath2vec model from the `"metapath2vec: Scalable Representation

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

Bases: *models.BaseModel*

The Metapath2vec model from the "metapath2vec: Scalable Representation Learning for Heterogeneous Networks" paper

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

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

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

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

_walk(*self*, *start_node*, *walk_length*, *schema=None*)

_simulate_walks(*self*, *walk_length*, *num_walks*, *schema='No'*)

models.emb.netmf

Module Contents

Classes

NetMF	The NetMF model from the `"Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec"

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

Bases: *models.BaseModel*

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

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

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

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

train (*self*, *G*)

_compute_deepwalk_matrix (*self*, *A*, *window*, *b*)

_approximate_normalized_laplacian (*self*, *A*, *rank*, *which='LA'*)

_deepwalk_filter (*self*, *evals*, *window*)

_approximate_deepwalk_matrix (*self*, *evals*, *D_rt_invU*, *window*, *vol*, *b*)

models.emb.netsmf

Module Contents

Classes

NetSMF	The NetSMF model from the `"NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization"

class models.emb.netsmf.**NetSMF** (*dimension*, *window_size*, *negative*, *num_round*, *worker*)
Bases: *models.BaseModel*

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

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

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

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

train (*self*, *G*)

_get_embedding_rand (*self*, *matrix*)

_path_sampling (*self*, *u*, *v*, *r*)

_random_walk_matrix (*self*, *pid*)

models.emb.node2vec

Module Contents

Classes

Node2vec	The node2vec model from the `"node2vec: Scalable feature learning for networks"

class models.emb.node2vec.**Node2vec**(*dimension*, *walk_length*, *walk_num*, *window_size*, *worker*, *iteration*, *p*, *q*)

Bases: *models.BaseModel*

The node2vec model from the "node2vec: Scalable feature learning for networks" paper

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

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

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

train(*self*, *G*)

_node2vec_walk(*self*, *walk_length*, *start_node*)

_simulate_walks(*self*, *num_walks*, *walk_length*)

_get_alias_edge(*self*, *src*, *dst*)

_preprocess_transition_probs(*self*)

models.emb.prone

Module Contents

Classes

ProNE	The ProNE model from the `"ProNE: Fast and Scalable Network Representation Learning"

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

Bases: *models.BaseModel*

The ProNE model from the "ProNE: Fast and Scalable Network Representation Learning" paper.

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

static add_args(*parser*)

Add model-specific arguments to the parser.

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

train(*self*, *G*)

_get_embedding_rand(*self*, *matrix*)

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

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

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

models.emb.pte

Module Contents

Classes

PTE	The PTE model from the ``"PTE: Predictive Text Embedding through Large-scale Heterogeneous Text Networks"

class models.emb.pte.**PTE**(*dimension*, *walk_length*, *walk_num*, *negative*, *batch_size*, *alpha*)
Bases: *models.BaseModel*

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

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

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

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

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

_update(*self*, *vec_u*, *vec_v*, *vec_error*, *label*)

_train_line(*self*)

models.emb.sdne

Module Contents

Classes

SDNE_layer	
SDNE	The SDNE model from the `"Structural Deep Network Embedding"

class models.emb.sdne.**SDNE_layer**(*num_node*, *hidden_size1*, *hidden_size2*, *droput*, *alpha*, *beta*, *nu1*, *nu2*)

Bases: torch.nn.Module

forward(*self*, *adj_mat*, *l_mat*)

get_emb(*self*, *adj*)

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

Bases: *models.BaseModel*

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

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

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

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

train(*self*, *G*)

models.emb.spectral

Module Contents

Classes

Spectral	The Spectral clustering model from the `"Leveraging social media networks for classication"

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

Bases: *models.BaseModel*

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

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

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

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

train(*self*, *G*)

models.nn

Submodules

models.nn.asgcn

Module Contents

Classes

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

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

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

reset_parameters(*self*)

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

__repr__(*self*)

class models.nn.asgcn.**ASGCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*, *sample_size*)
Bases: *models.BaseModel*

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

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

reset_parameters(*self*)

set_adj(*self*, *edge_index*, *num_nodes*)

compute_adjlist(*self*, *sp_adj*, *max_degree=32*)
Transfer sparse adjacent matrix to adj-list format

from_adjlist(*self*, *adj*)
Transfer adj-list format to sparsetensor

_sample_one_layer(*self*, *x*, *adj*, *v*, *sample_size*)

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

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

models.nn.dgl_gcc

Module Contents

Classes

NodeClassificationDataset	
GCC	

Functions

batcher()	
test_moco(train_loader, model, opt)	one epoch training for moco
eigen_decomposision(n, k, laplacian, hidden_size, retry)	
_add_undirected_graph_positional_embedding(g, hidden_size, retry=10)	
_rwr_trace_to_dgl_graph(g, seed, trace, positional_embedding_size, entire_graph=False)	

models.nn.dgl_gcc.**batcher**()

models.nn.dgl_gcc.**test_moco**(*train_loader*, *model*, *opt*)
one epoch training for moco

models.nn.dgl_gcc.**eigen_decomposision**(*n*, *k*, *laplacian*, *hidden_size*, *retry*)

models.nn.dgl_gcc.**_add_undirected_graph_positional_embedding**(*g*, *hidden_size*, *retry=10*)

models.nn.dgl_gcc.**_rwr_trace_to_dgl_graph**(*g*, *seed*, *trace*, *positional_embedding_size*, *entire_graph=False*)

class models.nn.dgl_gcc.**NodeClassificationDataset**(*data*, *rw_hops=64*, *subgraph_size=64*, *restart_prob=0.8*, *positional_embedding_size=32*, *step_dist=[1.0, 0.0, 0.0]*)

Bases: object

_create_dgl_graph(*self*, *data*)

__len__(*self*)

_convert_idx(*self*, *idx*)

__getitem__(*self*, *idx*)

class models.nn.dgl_gcc.**GCC**(*load_path*)
Bases: *models.BaseModel*

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

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

train(*self*, *data*)

models.nn.fastgcn

Module Contents

Classes

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

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

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

reset_parameters(*self*)

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

__repr__(*self*)

class models.nn.fastgcn.**FastGCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*, *sample_size*)
Bases: *models.BaseModel*

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

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

set_adj(*self*, *edge_index*, *num_nodes*)

_sample_one_layer(*self*, *sampled*, *sample_size*)

_generate_adj(*self*, *sample1*, *sample2*)

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

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

models.nn.gat

Module Contents

Classes

GraphAttentionLayer	Simple GAT layer, similar to https://arxiv.org/abs/1710.10903
SpecialSpmmFunction	Special function for only sparse region backpropataion layer.
SpecialSpmm	
SpGraphAttentionLayer	Sparse version GAT layer, similar to https://arxiv.org/abs/1710.10903
PetarVGAT	
PetarVSpGAT	

class models.nn.gat.**GraphAttentionLayer**(*in_features*, *out_features*, *dropout*, *alpha*, *concat=True*)

Bases: torch.nn.Module

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

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

__repr__(*self*)

class models.nn.gat.**SpecialSpmmFunction**

Bases: torch.autograd.Function

Special function for only sparse region backpropataion layer.

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

static backward(*ctx*, *grad_output*)

class models.nn.gat.**SpecialSpmm**

Bases: torch.nn.Module

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

class models.nn.gat.**SpGraphAttentionLayer**(*in_features*, *out_features*, *dropout*, *alpha*, *concat=True*)

Bases: torch.nn.Module

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

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

__repr__(*self*)

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

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

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

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.gcn

Module Contents

Classes

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

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

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

reset_parameters(*self*)

forward(*self*, *input*, *edge_index*)

__repr__(*self*)

class models.nn.gcn.**TKipfGCN**(*nfeat*, *nhid*, *nclass*, *dropout*)
Bases: *models.BaseModel*

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.graphsage

Module Contents

Classes

Graphsage

class models.nn.graphsage.**Graphsage**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *sample_size*, *dropout*)

Bases: *models.BaseModel*

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

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

sampler(*self*, *edge_index*, *num_sample*)

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.mixhop

Module Contents

Classes

MixHop

class models.nn.mixhop.**MixHop**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)

Bases: *models.BaseModel*

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.mlp

Module Contents

Classes

MLP	

class models.nn.mlp.**MLP**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.patchy_san

Module Contents

Classes

PatchySAN	The Patchy-SAN model from the `"Learning Convolutional Neural Networks for Graphs"

Functions

assemble_neighbor(G, node, num_neighbor, sorted_nodes)	assemble neighbors for node with BFS strategy
cmp(s1, s2)	
one_dim_wl(graph_list, init_labels, iteration=5)	1-dimension Wl method used for node normalization for all the subgraphs
node_selection_with_1d_wl(G, features, num_channel, num_sample, num_neighbor, stride)	construct features for cnn
get_single_feature(data, num_features, num_classes, num_sample, num_neighbor, stride=1)	construct features

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

Bases: *models.BaseModel*

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

Args: batch_size (int) : The batch size of training. sample (int) : Number of chosen vertexes. stride (int) : Node selection stride. neighbor (int) : The number of neighbor for each node. iteration (int) : The number of training iteration.

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

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

classmethod split_dataset(*self*, *dataset*, *args*)

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

forward(*self*, *batch*)

models.nn.patchy_san.**assemble_neighbor**(*G*, *node*, *num_neighbor*, *sorted_nodes*)
assemble neighbors for node with BFS strategy

models.nn.patchy_san.**cmp**(*s1*, *s2*)

models.nn.patchy_san.**one_dim_wl**(*graph_list*, *init_labels*, *iteration=5*)
1-dimension Wl method used for node normalization for all the subgraphs

models.nn.patchy_san.**node_selection_with_1d_wl**(*G*, *features*, *num_channel*, *num_sample*, *num_neighbor*, *stride*)
construct features for cnn

models.nn.patchy_san.**get_single_feature**(*data*, *num_features*, *num_classes*, *num_sample*, *num_neighbor*, *stride=1*)
construct features

models.nn.pyg_cheb

Module Contents

Classes

Chebyshev

class models.nn.pyg_cheb.**Chebyshev**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*, *filter_size*)
Bases: *models.BaseModel*

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_dgcnn

Module Contents

Classes

DGCNN	EdgeConv and DynamicGraph in paper `"Dynamic Graph CNN for Learning on

class models.nn.pyg_dgcnn.**DGCNN**(*in_feats*, *hidden_dim*, *out_feats*, *k=20*, *dropout=0.5*)

Bases: *models.BaseModel*

EdgeConv and DynamicGraph in paper *"Dynamic Graph CNN for Learning on Point Clouds" <https://arxiv.org/pdf/1801.07829.pdf>__* .

in_feats [int] Size of each input sample.

out_feats [int] Size of each output sample.

hidden_dim [int] Dimension of hidden layer embedding.

k [int] Number of neareast neighbors.

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

classmethod split_dataset(*cls*, *dataset*, *args*)

forward(*self*, *batch*)

models.nn.pyg_diffpool

Module Contents

Classes

EntropyLoss	
LinkPredLoss	
GraphSAGE	GraphSAGE from "Inductive Representation Learning on Large Graphs".
BatchedGraphSAGE	GraphSAGE with mini-batch
BatchedDiffPoolLayer	DIFFPOOL from paper `"Hierarchical Graph Representation Learning
BatchedDiffPool	DIFFPOOL layer with batch forward
DiffPool	DIFFPOOL from paper `Hierarchical Graph Representation Learning

Functions

toBatchedGraph(batch_adj, batch_feat, node_per_pool_graph)	

class models.nn.pyg_diffpool.**EntropyLoss**

Bases: torch.nn.Module

forward(*self*, *adj*, *anext*, *s_l*)

class models.nn.pyg_diffpool.**LinkPredLoss**

Bases: torch.nn.Module

forward(*self*, *adj*, *anext*, *s_l*)

class models.nn.pyg_diffpool.**GraphSAGE**(*in_feats*, *hidden_dim*, *out_feats*, *num_layers*, *dropout=0.5*, *normalize=False*, *concat=False*, *use_bn=False*)

Bases: torch.nn.Module

GraphSAGE from "Inductive Representation Learning on Large Graphs".

..math:: h^{i+1}_{mathcal{N}(v)}=AGGREGATE_{k}(h_{u}^{k}) h^{k+1}_{v} = sigma(mathbf{W}^{k}·CONCAT(h_{v}^{k}, h_{mathcal{N}(v)}))

Args: in_feats (int) : Size of each input sample. hidden_dim (int) : Size of hidden layer dimension. out_feats (int) : Size of each output sample. num_layers (int) : Number of GraphSAGE Layers. dropout (float, optional) : Size of dropout, default: 0.5. normalize (bool, optional) : Normalze features after each layer if True, default: True.

forward(*self*, *x*, *edge_index*, *edge_weight=None*)

class models.nn.pyg_diffpool.**BatchedGraphSAGE**(*in_feats*, *out_feats*, *use_bn=True*, *self_loop=True*)

Bases: torch.nn.Module

GraphSAGE with mini-batch

Args: in_feats (int) : Size of each input sample. out_feats (int) : Size of each output sample. use_bn (bool) : Apply batch normalization if True, default: True. self_loop (bool) : Add self loop if True, default: True.

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

class models.nn.pyg_diffpool.**BatchedDiffPoolLayer**(*in_feats*, *out_feats*, *assign_dim*, *batch_size*, *dropout=0.5*, *link_pred_loss=True*, *entropy_loss=True*)

Bases: torch.nn.Module

DIFFPOOL from paper "Hierarchical Graph Representation Learning with Differentiable Pooling".

$$X^{(l+1)} = S^{l)T} Z^{(l)} A^{(l+1)} = S^{(l)T} A^{(l)} S^{(l)} Z^{(l)} = GNN_{l,embed}(A^{(l)}, X^{(l)}) S^{(l)} = softmax(GNN_{l,pool}(A^{(l)}, X^{(l)}))$$

in_feats [int] Size of each input sample.

out_feats [int] Size of each output sample.

assign_dim [int] Size of next adjacency matrix.

batch_size [int] Size of each mini-batch.

dropout [float, optional] Size of dropout, default: 0.5.

link_pred_loss [bool, optional] Use link prediction loss if True, default: `True`.

forward(*self*, *x*, *edge_index*, *batch*, *edge_weight=None*)

get_loss(*self*)

class models.nn.pyg_diffpool.**BatchedDiffPool**(*in_feats*, *next_size*, *emb_size*, *use_bn=True*, *self_loop=True*, *use_link_loss=False*, *use_entropy=True*)

Bases: `torch.nn.Module`

DIFFPOOL layer with batch forward

in_feats [int] Size of each input sample.

next_size [int] Size of next adjacency matrix.

emb_size [int] Dimension of next node feature matrix.

use_bn [bool, optional] Apply batch normalization if True, default: `True`.

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

use_link_loss [bool, optional] Use link prediction loss if True, default: `True`.

use_entropy [bool, optioinal] Use entropy prediction loss if True, default: `True`.

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

get_loss(*self*)

models.nn.pyg_diffpool.**toBatchedGraph**(*batch_adj*, *batch_feat*, *node_per_pool_graph*)

class models.nn.pyg_diffpool.**DiffPool**(*in_feats*, *hidden_dim*, *embed_dim*, *num_classes*, *num_layers*, *num_pool_layers*, *assign_dim*, *pooling_ratio*, *batch_size*, *dropout=0.5*, *no_link_pred=True*, *concat=False*, *use_bn=False*)

Bases: *models.BaseModel*

DIFFPOOL from paper Hierarchical Graph Representation Learning with Differentiable Pooling.

in_feats [int] Size of each input sample.

hidden_dim [int] Size of hidden layer dimension of GNN.

embed_dim [int] Size of 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.

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

classmethod split_dataset(*cls*, *dataset*, *args*)

reset_parameters(*self*)

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

forward(*self*, *batch*)

loss(*self*, *prediction*, *label*)

models.nn.pyg_drgat

Module Contents

Classes

DrGAT

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

Bases: *models.BaseModel*

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_drgcn

Module Contents

Classes

DrGCN

class models.nn.pyg_drgcn.**DrGCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)

Bases: *models.BaseModel*

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

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

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_gat

Module Contents

Classes

GAT

class models.nn.pyg_gat.**GAT**(*num_features*, *num_classes*, *hidden_size*, *num_heads*, *dropout*)

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_gcn

Module Contents

Classes

GCN

class models.nn.pyg_gcn.**GCN**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_gin

Module Contents

Classes

GINLayer	Graph Isomorphism Network layer from paper `"How Powerful are Graph
GINMLP	Multilayer perception with batch normalization
GIN	Graph Isomorphism Network from paper `"How Powerful are Graph

class models.nn.pyg_gin.**GINLayer**(*apply_func=None*, *eps=0*, *train_eps=True*)

Bases: torch.nn.Module

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

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

apply_func [callable layer function)] layer or function applied to update node feature

eps [float32, optional] Initial *epsilon* value.

train_eps [bool, optional] If True, *epsilon* will be a learnable parameter.

forward(*self*, *x*, *edge_index*, *edge_weight=None*)

class models.nn.pyg_gin.**GINMLP**(*in_feats*, *out_feats*, *hidden_dim*, *num_layers*, *use_bn=True*, *activation=None*)

Bases: torch.nn.Module

Multilayer perception with batch normalization

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

in_feats [int] Size of each input sample.

out_feats [int] Size of each output sample.

hidden_dim [int] Size of hidden layer dimension.

use_bn [bool, optional] Apply batch normalization if True, default: `True).

forward(*self*, *x*)

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

Bases: *models.BaseModel*

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

Args:

num_layers [int] Number of GIN layers

in_feats [int] Size of each input sample

out_feats [int] Size of each output sample

hidden_dim [int] Size of each hidden layer dimension

num_mlp_layers [int] Number of MLP layers

eps [float32, optional] Initial *epsilon* value, default: `0`

pooling [str, optional] Aggregator type to use, default:`sum`

train_eps [bool, optional] If True, *epsilon* will be a learnable parameter, default: `True`

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

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

classmethod split_dataset(*cls*, *dataset*, *args*)

forward(*self*, *batch*)

loss(*self*, *output*, *label=None*)

models.nn.pyg_gtn

Module Contents

Classes

GTConv
GTLayer
GTN

class models.nn.pyg_gtn.**GTConv**(*in_channels*, *out_channels*, *num_nodes*)
Bases: `torch.nn.Module`

reset_parameters(*self*)

forward(*self*, *A*)

class models.nn.pyg_gtn.**GTLayer**(*in_channels*, *out_channels*, *num_nodes*, *first=True*)
Bases: `torch.nn.Module`

forward(*self*, *A*, *H_=None*)

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

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

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

normalization(*self*, *H*)

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

forward(*self*, *A*, *X*, *target_x*, *target*)

loss(*self*, *data*)

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

models.nn.pyg_han

Module Contents

Classes

AttentionLayer	
HANLayer	
HAN	

class models.nn.pyg_han.**AttentionLayer**(*num_features*)
Bases: torch.nn.Module

forward(*self*, *x*)

class models.nn.pyg_han.**HANLayer**(*num_edge*, *w_in*, *w_out*)
Bases: torch.nn.Module

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

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

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

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

forward(*self*, *A*, *X*, *target_x*, *target*)

loss(*self*, *data*)

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

models.nn.pyg_infograph

Module Contents

Classes

SUPEncoder	Encoder used in supervised model with Set2set in paper `"Order Matters: Sequence to sequence for sets"
Encoder	Encoder stacked with GIN layers
FF	Residual MLP layers.

continues on next page

Table 88 – continued from previous page

InfoGraph	Implementation of Infograph in paper `"InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation

class models.nn.pyg_infograph.**SUPEncoder**(*num_features*, *dim*, *num_layers=1*)

Bases: torch.nn.Module

Encoder used in supervised model with Set2set in paper *"Order Matters: Sequence to sequence for sets" <https://arxiv.org/abs/1511.06391>* and NNConv in paper *"Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs" <https://arxiv.org/abs/1704.02901>*

forward(*self*, *x*, *edge_index*, *batch*, *edge_attr*)

class models.nn.pyg_infograph.**Encoder**(*in_feats*, *hidden_dim*, *num_layers=3*, *num_mlp_layers=2*, *pooling='sum'*)

Bases: torch.nn.Module

Encoder stacked with GIN layers

in_feats [int] Size of each input sample.

hidden_feats [int] Size of output embedding.

num_layers [int, optional] Number of GIN layers, default: 3.

num_mlp_layers [int, optional] Number of MLP layers for each GIN layer, default: 2.

pooling [str, optional] Aggragation type, default : sum.

forward(*self*, *x*, *edge_index*, *batch*, **args*)

class models.nn.pyg_infograph.**FF**(*in_feats*, *out_feats*)

Bases: torch.nn.Module

Residual MLP layers.

..math:: out = mathbf{MLP}(x) + mathbf{Linear}(x)

in_feats [int] Size of each input sample

out_feats [int] Size of each output sample

forward(*self*, *x*)

class models.nn.pyg_infograph.**InfoGraph**(*in_feats*, *hidden_dim*, *out_feats*, *num_layers=3*, *unsup=True*)

Bases: *models.BaseModel*

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

classmethod split_dataset(*cls*, *dataset*, *args*)

reset_parameters(*self*)

forward(*self*, *batch*)

sup_forward(*self*, *x*, *edge_index=None*, *batch=None*, *label=None*, *edge_attr=None*)

unsup_forward(*self*, *x*, *edge_index=None*, *batch=None*)

sup_loss(*self*, *prediction*, *label=None*)

unsup_loss(*self*, *x*, *edge_index=None*, *batch=None*)

unsup_sup_loss(*self*, *x*, *edge_index*, *batch*)

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

models.nn.pyg_infomax

Module Contents

Classes

Encoder	
Infomax	

Functions

corruption(x, edge_index)	

class models.nn.pyg_infomax.**Encoder**(*in_channels*, *hidden_channels*)

Bases: torch.nn.Module

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

models.nn.pyg_infomax.**corruption**(*x*, *edge_index*)

class models.nn.pyg_infomax.**Infomax**(*num_features*, *num_classes*, *hidden_size*)

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_sortpool

Module Contents

Classes

SortPool	Implementation of sortpooling in paper `"An End-to-End Deep Learning

Functions

scatter_sum(src, index, dim, dim_size)

spare2dense_batch(x, batch=None, fill_value=0)

models.nn.pyg_sortpool.**scatter_sum**(*src*, *index*, *dim*, *dim_size*)

models.nn.pyg_sortpool.**spare2dense_batch**(*x*, *batch=None*, *fill_value=0*)

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

Bases: *models.BaseModel*

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

in_feats [int] Size of each input sample.

out_feats [int] Size of each output sample.

hidden_dim [int] Dimension of hidden layer embedding.

num_classes [int] Number of target classes.

num_layers [int] Number of graph neural network layers before pooling.

k [int, optional] Number of selected features to sort, default: `30`.

out_channel [int] Number of the first convolution's output channels.

kernel_size [int] Size of the first convolution's kernel.

dropout [float, optional] Size of dropout, default: `0.5`.

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

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

classmethod split_dataset(*cls*, *dataset*, *args*)

forward(*self*, *batch*)

models.nn.pyg_srgcn

Module Contents

Classes

NodeAdaptiveEncoder
SrgcnHead
SrgcnSoftmaxHead
SRGCN

class models.nn.pyg_srgcn.**NodeAdaptiveEncoder**(*num_features*, *dropout=0.5*)

Bases: nn.Module

forward(*self*, *x*)

class models.nn.pyg_srgcn.**SrgcnHead**(*num_features*, *out_feats*, *attention*, *activation*, *normalization*, *nhop*, *subheads=2*, *dropout=0.5*, *node_dropout=0.5*, *alpha=0.2*, *concat=True*)

Bases: nn.Module

forward(*self*, *x*, *edge_index*, *edge_attr*)

class models.nn.pyg_srgcn.**SrgcnSoftmaxHead**(*num_features*, *out_feats*, *attention*, *activation*, *nhop*, *normalization*, *dropout=0.5*, *node_dropout=0.5*, *alpha=0.2*)

Bases: nn.Module

forward(*self*, *x*, *edge_index*, *edge_attr*)

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

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

forward(*self*, *batch*)

loss(*self*, *data*)

predict(*self*, *data*)

models.nn.pyg_unet

Module Contents

Classes

UNet

class models.nn.pyg_unet.**UNet**(*num_features*, *num_classes*, *hidden_size*, *num_layers*, *dropout*)

Bases: *models.BaseModel*

static add_args(*parser*)

Add model-specific arguments to the parser.

classmethod build_model_from_args(*cls*, *args*)

Build a new model instance.

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

loss(*self*, *data*)

predict(*self*, *data*)

6.7.2 Submodules

models.base_model

Module Contents

Classes

BaseModel

class models.base_model.**BaseModel**

Bases: torch.nn.Module

static add_args(*parser*)

Add model-specific arguments to the parser.

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

Build a new model instance.

6.7.3 Package Contents

Classes

BaseModel	

Functions

register_model(name)	New model types can be added to cogdl with the *register_model()*
alias_setup(probs)	Compute utility lists for non-uniform sampling from discrete distributions.
alias_draw(J, q)	Draw sample from a non-uniform discrete distribution using alias sampling.
build_model(args)	

class models.**BaseModel**

Bases: torch.nn.Module

static add_args(*parser*)

Add model-specific arguments to the parser.

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

Build a new model instance.

models.**pyg = False**

models.**dgl_import = False**

models.**MODEL_REGISTRY**

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

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

For example:

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

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

models.**alias_setup**(*probs*)

Compute utility lists for non-uniform sampling from discrete distributions. Refer to https://hips.seas.harvard.edu/blog/2013/03/03/the-alias-method-efficient-sampling-with-many-discrete-outcomes/ for details

models.**alias_draw**(*J*, *q*)

Draw sample from a non-uniform discrete distribution using alias sampling.

models.**model_name**

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

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

U

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Y