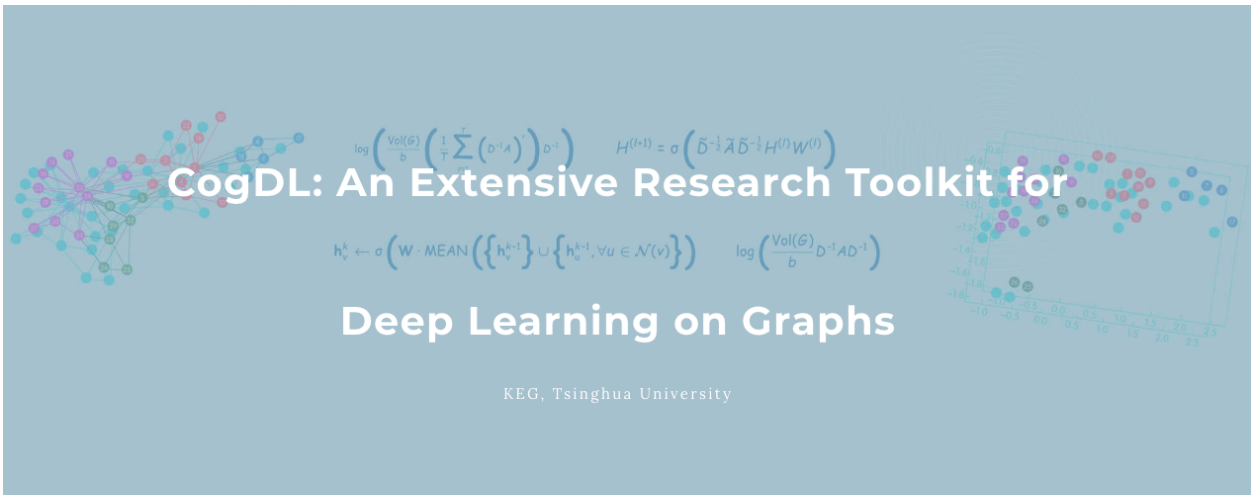
CogDL Documentation

Release 0.2.0

KEG

Mar 03, 2021

Get Started



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
  - Graph pre-training
  - Graph clustering
  - Graph similarity search

CHAPTER 1

Install

- Python version >= 3.6
- PyTorch version >= 1.6.0
- PyTorch Geometric (recommended)
- Deep Graph Library (optional)

Please follow the instructions here to install PyTorch: https://github.com/pytorch/pytorch#installation, PyTorch Geometric https://github.com/rusty1s/pytorch_geometric/#installation and Deep Graph Library https://docs.dgl.ai/install/index.html.

Install cogdl with other dependencies:

```
pip install cogdl
```

If you want to experiment with the latest CogDL features which did not get released yet, you can install CogDL via:

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

CHAPTER 2

Quick Start

2.1 API Usage

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

```
from cogdl import experiment

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

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

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

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

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

2.2 Command-Line Usage

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

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

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

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

Expected output:

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

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

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

Expected output:

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

CHAPTER 3

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.

Semi-supervied Node Classification Methods

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

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

First we define the *NodeClassification* class.

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

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

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

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

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

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

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

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

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

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

def _test_step(self, split="val"):
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

To run NodeClassification, we can use the following command:

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

Then We get experimental results like this:

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

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* or *Accuracy* to measure the performance.

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

Unsupervised Graph Embedding Methods

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

Unsupervised Graph Neural Network Representation Learning Methods

Method	Sampling	Inductive	Reproducibility
DGI			
MVGRL			
GRACE			
GraphSAGE			

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

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

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

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

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

```
dataset = build_dataset(args)
self.data = dataset[0]
if issubclass(dataset.__class__.__bases__[0], InMemoryDataset):
```

(continues on next page)

(continued from previous page)

```
    self.num_nodes = self.data.y.shape[0]
    self.num_classes = dataset.num_classes
    self.label_matrix = np.zeros((self.num_nodes, self.num_classes), dtype=int)
    self.label_matrix[range(self.num_nodes), self.data.y] = 1
    self.data.edge_attr = self.data.edge_attr.t()
else:
    self.label_matrix = self.data.y
    self.num_nodes, self.num_classes = self.data.y.shape
```

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

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


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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

```
from sklearn.multiclass import OneVsRestClassifier

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

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

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

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

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

To run UnsupervisedNodeClassification, we can use following instruction:

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

Then We get experimental results like this:

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

3.3 Supervised Graph Classification

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

** Supervised Graph Classification Methods **

Method	Node Feature	Kernel	Reproducibility
GIN			
DiffPool			
SortPool			
PATCH_SAN			
DGCNN			
SAGPool			

Task Design

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

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

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

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

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

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

def _train_step(self):
    self.model.train()
    loss_n = 0
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
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".

Unsupervised Graph Classificaton Methods

Method	Node Feature	Kernel	Reproducibility
InfoGraph			
DGK			
Graph2Vec			
HGP_SL			

Task Design

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

```
@register_task("unsupervised_graph_classification")
class UnsupervisedGraphClassification(BaseTask):
    r"""Unsupervised graph classification"""
    @staticmethod
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
            clf.fit(X_train, y_train)
            preds = clf.predict(X_test)
            # ...
```
```

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

**Create a model**

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

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

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

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

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

Run

To run UnsupervisedGraphClassification, we can use the following command:

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

Then we get experimental results like this:

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

3.5 Link Prediction

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

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

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

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

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

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

3.6 Other Tasks

Heterogeneous Graph Embedding Methods

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

Pretrained Graph Models

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

3.7 Create new tasks

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

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

(continues on next page)

(continued from previous page)

```
    def __init__(self, args):
        pass

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

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

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

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

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

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

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

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

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

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

```
from cogdl.tasks import build_task
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

CHAPTER 4

Trainer

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

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

Design

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

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

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

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

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

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

    # preprocess data
    self.train_loader = NeighborSampler(
        data=self.data,
        mask=self.data.train_mask,
```

(continues on next page)

(continued from previous page)

```
        sizes=self.sample_size,
        batch_size=self.batch_size,
        num_workers=self.num_workers,
        shuffle=True,
    )
    self.test_loader = NeighborSampler(
        data=self.data, mask=None, sizes=[-1], batch_size=self.batch_size,␣
↪shuffle=False
    )
    # move model to GPU
    self.model = model.to(self.device)

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

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

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

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

CHAPTER 5

Model

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

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

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

Then we use function decorator to declare new model for CogDL

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

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

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

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

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

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

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

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

CHAPTER 6

Dataset

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

```
@register_dataset("ppi")
class PPIDataset(MatlabMatrix):
    def __init__(self):
        dataset, filename = "ppi", "Homo_sapiens"
        url = "http://snap.stanford.edu/node2vec/"
        path = osp.join(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. More implemented datasets are at https://github.com/THUDM/cogdl/tree/master/cogdl/datasets.

CHAPTER 7

data

CHAPTER 8

datasets

CHAPTER 9

tasks

CHAPTER 10

models

CHAPTER 11

layers

CHAPTER 12

options

CHAPTER 13

utils

CHAPTER 14

experiments

CHAPTER 15

pipelines

CHAPTER 16

Indices and tables

- genindex
- modindex
- search