DeepRobust

Release 0.1.1

May 21, 2021

Installation

DeepRobust is a pytorch adversarial learning library, which contains most popular attack and defense algorithms in image domain and graph domain.

CHAPTER 1

Installation

1. Activate your virtual environment
2. Install package

   Install the newest deeprobust:

   ```
   git clone https://github.com/DSE-MSU/DeepRobust.git
   cd DeepRobust
   python setup.py install
   ```

   Or install via pip (may not contain all the new features)

   ```
   pip install deeprobust
   ```

Note: If you meet any installation problem, feel free to open an issue in the our github page

CHAPTER 2

Graph Dataset

We briefly introduce the dataset format of DeepRobust through self-contained examples. In essence, DeepRobust-Graph provides the following main features:

2.1 Clean (Unattacked) Graphs for Node Classification

Graphs are ubiquitous data structures describing pairwise relations between entities. A single clean graph in DeepRobust is described by an instance of `deeprobust.graph.data.Dataset`, which holds the following attributes by default:

- `data.adj`: Graph adjacency matrix in scipy.sparse.csr_matrix format with shape `[num_nodes, num_nodes]`
- `data.features`: Node feature matrix with shape `[num_nodes, num_node_features]`
- `data.labels`: Target to train against (may have arbitrary shape), *e.g.*, node-level targets of shape `[num_nodes, *]`
- `data.train_idx`: Array of training node indices
- `data.val_idx`: Array of validation node indices
- `data.test_idx`: Array of test node indices

By default, the loaded *deeprobust.graph.data.Dataset* will select the largest connect component of the graph, but users specify different settings by giving different parameters.

Currently DeepRobust supports the following datasets: `Cora`, `Cora-ML`, `Citeseer`, `Pubmed`, `Polblogs`, `ACM`, `BlogCatalog`, `Flickr`, `UAI`. More details about the datasets can be found here.

By default, the data splits are generated by `deeprobust.graph.utils.get_train_val_test`, which randomly split the data into 10%/10%/80% for training/validaiton/test. You can also generate splits by yourself by using `deeprobust.graph.utils.get_train_val_test` or `deeprobust.graph.utils.get_train_val_test_gcn`. It is worth noting that there is parameter `setting` that can be passed into this class. It can be chosen from *["nettack", "gcn", "prognn"]*:

- `setting="nettack"`: the data splits are 10%/10%/80% and using the largest connected component of the graph;
- `setting="gcn"`: use the full graph and the data splits will be: 20 nodes per class for training, 500 nodes for validation and 1000 nodes for testing (randomly choosen);
- `setting="prognn"`: use the largest connected component and the data splits are provided by ProGNN (10%/10%/80%);

Note: The 'netack' and 'gcn' setting do not provide fixed split, i.e., different random seed would return different data splits.

Note: If you hope to use the full graph, please use the 'gcn' setting.

The following example shows how to load DeepRobust datasets

```
from deeprobust.graph.data import Dataset
# loading cora dataset
data = Dataset(root='/tmp/', name='cora', seed=15)
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
# you can also split the data by yourself
idx_train, idx_val, idx_test = get_train_val_test(adj.shape[0], val_size=0.1, test_
↪size=0.8)

# loading acm dataset
data = Dataset(root='/tmp/', name='acm', seed=15)
```

DeepRobust also provides access to Amazon and Coauthor datasets loaded from Pytorch Geometric: `Amazon-Computers`, `Amazon-Photo`, `Coauthor-CS`, `Coauthor-Physics`.

Users can also easily create their own datasets by creating a class with the following attributes: `data.adj`, `data.features`, `data.labels`, `data.train_idx`, `data.val_idx`, `data.test_idx`.

2.2 Attacked Graphs for Node Classification

DeepRobust provides the attacked graphs perturbed by metattack and nettack. The graphs are attacked using authors' Tensorflow implementation, on random split using seed 15. The download link can be found in ProGNN code and the performance of various GNNs can be found in ProGNN paper. They are instances of *deeprobust.graph.data.PrePtbDataset* with only one attribute `adj`. Hence, *deeprobust.graph.data.PrePtbDataset* is often used together with *deeprobust.graph.data.Dataset* to obtain node features and labels.

For metattack, DeepRobust provides attacked graphs for Cora, Citeseer, Polblogs and Pubmed, and the perturbation rate can be chosen from [0.05, 0.1, 0.15, 0.2, 0.25].

```
from deeprobust.graph.data import Dataset, PrePtbDataset
# You can either use setting='prognn' or seed=15 to get the prognn splits
data = Dataset(root='/tmp/', name='cora', setting='prognn')
data = Dataset(root='/tmp/', name='cora', seed=15) # since the attacked graph are␣
↪generated under seed 15
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
# Load meta attacked data
perturbed_data = PrePtbDataset(root='/tmp/',
                                        name='cora',
                                        attack_method='meta',
                                        ptb_rate=0.05)
perturbed_adj = perturbed_data.adj
```

For nettack, DeepRobust provides attacked graphs for Cora, Citeseer, Polblogs and Pubmed, and ptb_rate indicates the number of perturbations made on each node. It can be chosen from [1.0, 2.0, 3.0, 4.0, 5.0].

```
from deeprobust.graph.data import Dataset, PrePtbDataset
# data = Dataset(root='/tmp/', name='cora', seed=15) # since the attacked graph are␣
↪generated under seed 15
data = Dataset(root='/tmp/', name='cora', setting='prognn')
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
# Load nettack attacked data
perturbed_data = PrePtbDataset(root='/tmp/', name='cora',
                                        attack_method='nettack',
                                        ptb_rate=3.0) # here ptb_rate means number of␣
↪perturbation per nodes
perturbed_adj = perturbed_data.adj
idx_test = perturbed_data.target_nodes
```

2.3 Converting Graph Data between DeepRobust and PyTorch Geometric

Given the popularity of PyTorch Geometric in the graph representation learning community, we also provide tools for converting data between DeepRobust and PyTorch Geometric. We can use *deeprobust.graph.data.Dpr2Pyg* to convert DeepRobust data to PyTorch Geometric and use *deeprobust.graph.data.Pyg2Dpr* to convert Pytorch Geometric data to DeepRobust. For example, we can first create an instance of the Dataset class and convert it to pytorch geometric data format.

```
from deeprobust.graph.data import Dataset, Dpr2Pyg, Pyg2Dpr
data = Dataset(root='/tmp/', name='cora') # load clean graph
pyg_data = Dpr2Pyg(data) # convert dpr to pyg
print(pyg_data)
print(pyg_data[0])
dpr_data = Pyg2Dpr(pyg_data) # convert pyg to dpr
print(dpr_data.adj)
```

2.4 Load OGB Datasets

Open Graph Benchmark (OGB) has provided various benchmark datasets. DeepRobsut now provides interface to convert OGB dataset format (Pyg data format) to DeepRobust format.

```
from ogb.nodeproppred import PygNodePropPredDataset
from deeprobust.graph.data import Pyg2Dpr
pyg_data = PygNodePropPredDataset(name = 'ogbn-arxiv')
dpr_data = Pyg2Dpr(pyg_data) # convert pyg to dpr
```

2.5 Load Pytorch Geometric Amazon and Coauthor Datasets

DeepRobust also provides access to the Amazon datasets and Coauthor datasets, i.e., *Amazon-Computers*, *Amazon-Photo*, *Coauthor-CS*, *Coauthor-Physics*, from Pytorch Geometric. Specifically, users can access them through *deeprobust.graph.data.AmazonPyg* and *deeprobust.graph.data.CoauthorPyg*. For example, we can directly load Amazon dataset from deeprobust in the format of pyg as follows,

```
from deeprobust.graph.data import AmazonPyg
computers = AmazonPyg(root='/tmp', name='computers')
print(computers)
print(computers[0])
photo = AmazonPyg(root='/tmp', name='photo')
print(photo)
print(photo[0])
```

Similarly, we can also load Coauthor dataset,

```
from deeprobust.graph.data import CoauthorPyg
cs = CoauthorPyg(root='/tmp', name='cs')
print(cs)
print(cs[0])
physics = CoauthorPyg(root='/tmp', name='physics')
print(physics)
print(physics[0])
```

CHAPTER 3

Introduction to Graph Attack with Examples

In this section, we introduce the graph attack algorithms provided in DeepRobust. Speficailly, they can be divied into two types: (1) targeted attack `deeprobust.graph.targeted_attack` and (2) global attack `deeprobust.graph.global_attack`.

3.1 Global (Untargeted) Attack for Node Classification

Global (untargeted) attack aims to fool GNNs into giving wrong predictions on all given nodes. Specifically, DeepRobust provides the following targeted attack algorithms:

- `deeprobust.graph.global_attack.Metattack`
- `deeprobust.graph.global_attack.MetaApprox`
- `deeprobust.graph.global_attack.DICE`
- `deeprobust.graph.global_attack.MinMax`
- `deeprobust.graph.global_attack.PGDAttack`
- `deeprobust.graph.global_attack.NIPA`
- `deeprobust.graph.global_attack.Random`
- `deeprobust.graph.global_attack.NodeEmbeddingAttack`
- `deeprobust.graph.global_attack.OtherNodeEmbeddingAttack`

All the above attacks except *NodeEmbeddingAttack* and *OtherNodeEmbeddingAttack* (see details here) take the adjacency matrix, node feature matrix and labels as input. Usually, the adjacency matrix is in the format of `scipy.sparse.csr_matrix` and feature matrix can either be `scipy.sparse.csr_matrix` or `numpy.array`. The attack algorithm will then transfer them into `torch.tensor` inside the class. It is also fine if you provide `torch.tensor` as input, since the algorithm can automatically deal with it. Now let's take a look at an example:

```
import numpy as np
from deeprobust.graph.data import Dataset
from deeprobust.graph.defense import GCN
from deeprobust.graph.global_attack import Metattack
data = Dataset(root='/tmp/', name='cora')
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
idx_unlabeled = np.union1d(idx_val, idx_test)
idx_unlabeled = np.union1d(idx_val, idx_test)
# Setup Surrogate model
surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu').
↪to('cpu')
surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
# Setup Attack Model
model = Metattack(surrogate, nnodes=adj.shape[0], feature_shape=features.shape,
        attack_structure=True, attack_features=False, device='cpu', lambda_=0).to('cpu
↪')
# Attack
model.attack(features, adj, labels, idx_train, idx_unlabeled, n_perturbations=10, ll_
↪constraint=False)
modified_adj = model.modified_adj # modified_adj is a torch.tensor
```

3.2 Targeted Attack for Node Classification

Targeted attack aims to fool GNNs into give wrong predictions on a subset of nodes. Specifically, DeepRobust provides the following targeted attack algorithms:

- *deeprobust.graph.targeted_attack.Nettack*
- *deeprobust.graph.targeted_attack.RLS2V*
- *deeprobust.graph.targeted_attack.FGA*
- *deeprobust.graph.targeted_attack.RND*
- *deeprobust.graph.targeted_attack.IGAttack*

All the above attacks take the adjacency matrix, node feature matrix and labels as input. Usually, the adjacency matrix is in the format of `scipy.sparse.csr_matrix` and feature matrix can either be `scipy.sparse.csr_matrix` or `numpy.array`. Now let's take a look at an example:

```
from deeprobust.graph.data import Dataset
from deeprobust.graph.defense import GCN
from deeprobust.graph.targeted_attack import Nettack
data = Dataset(root='/tmp/', name='cora')
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
# Setup Surrogate model
surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu').
↪to('cpu')
```

(continues on next page)

(continued from previous page)

```
surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
# Setup Attack Model
target_node = 0
model = Nettack(surrogate, nnodes=adj.shape[0], attack_structure=True, attack_
↪features=True, device='cpu').to('cpu')
# Attack
model.attack(features, adj, labels, target_node, n_perturbations=5)
modified_adj = model.modified_adj # scipy sparse matrix
modified_features = model.modified_features # scipy sparse matrix
```

Note that we also provide scripts in `test_nettack.py` for selecting nodes as reported in the nettack paper: (1) the 10 nodes with highest margin of classification, i.e. they are clearly correctly classified, (2) the 10 nodes with lowest margin (but still correctly classified) and (3) 20 more nodes randomly.

3.3 More Examples

More examples can be found in *deeprobust.graph.targeted_attack* and *deeprobust.graph.global_attack*. You can also find examples in github code examples and more details in attacks table.

CHAPTER 4

Introduction to Graph Defense with Examples

In this section, we introduce the graph attack algorithms provided in DeepRobust.

4.1 Test your model's robustness on poisoned graph

DeepRobust provides a series of defense methods that aim to enhance the robustness of GNNs.

Victim Models:

- *deeprobust.graph.defense.GCN*
- *deeprobust.graph.defense.GAT*
- *deeprobust.graph.defense.ChebNet*
- *deeprobust.graph.defense.SGC*

Node Embedding Victim Models: (see more details here)

- *deeprobust.graph.defense.DeepWalk*
- *deeprobust.graph.defense.Node2Vec*

Defense Methods:

- *deeprobust.graph.defense.GCNJaccard*
- *deeprobust.graph.defense.GCNSVD*
- *deeprobust.graph.defense.ProGNN*
- *deeprobust.graph.defense.RGCN*

- *deeprobust.graph.defense.SimPGCN*
- deeprobust.graph.defense.AdvTraining

1. Load pre-attacked graph data

```
from deeprobust.graph.data import Dataset, PrePtbDataset
# load the prognn splits by using setting='prognn'
# because the attacked graphs are generated under prognn splits
data = Dataset(root='/tmp/', name='cora', setting='prognn')

adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
# Load meta attacked data
perturbed_data = PrePtbDataset(root='/tmp/',
                    name='cora',
                    attack_method='meta',
                    ptb_rate=0.05)
perturbed_adj = perturbed_data.adj
```

2. You can also choose to load graphs attacked by nettack. See details here

```
# Load nettack attacked data
perturbed_data = PrePtbDataset(root='/tmp/', name='cora',
        attack_method='nettack',
        ptb_rate=3.0) # here ptb_rate means number of perturbation per␣
↪nodes
perturbed_adj = perturbed_data.adj
idx_test = perturbed_data.target_nodes
```

3. Train a victim model (GCN) on clearn/poinsed graph

```
from deeprobust.graph.defense import GCN
gcn = GCN(nfeat=features.shape[1],
    nhid=16,
    nclass=labels.max().item() + 1,
    dropout=0.5, device='cpu')
gcn = gcn.to('cpu')
gcn.fit(features, adj, labels, idx_train, idx_val) # train on clean graph␣
↪with earlystopping
gcn.test(idx_test)

gcn.fit(features, perturbed_adj, labels, idx_train, idx_val) # train on␣
↪poisoned graph
gcn.test(idx_test)
```

4. Train defense models (GCN-Jaccard, RGCN, ProGNN) poinsed graph

```
from deeprobust.graph.defense import GCNJaccard
model = GCNJaccard(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu').to('cpu')
model.fit(features, perturbed_adj, labels, idx_train, idx_val,␣
↪threshold=0.03)
model.test(idx_test)
```

```
from deeprobust.graph.defense import GCNJaccard
model = RGCN(nnodes=perturbed_adj.shape[0], nfeat=features.shape[1],
             nclass=labels.max()+1, nhid=32, device='cpu')
model.fit(features, perturbed_adj, labels, idx_train, idx_val,
          train_iters=200, verbose=True)
model.test(idx_test)
```

For details in training ProGNN, please refer to this page.

4.2 More Examples

More examples can be found in *deeprobust.graph.defense*. You can also find examples in github code examples and more details in defense table.

CHAPTER 5

Using PyTorch Geometric in DeepRobust

DeepRobust now provides interface to convert the data between PyTorch Geometric and DeepRobust.

Note: Before we start, make sure you have successfully installed torch_geometric. After you install torch_geometric, please reinstall DeepRobust to activate the following functions.

5.1 Converting Graph Data between DeepRobust and PyTorch Geometric

Given the popularity of PyTorch Geometric in the graph representation learning community, we also provide tools for converting data between DeepRobust and PyTorch Geometric. We can use *deeprobust.graph.data.Dpr2Pyg* to convert DeepRobust data to PyTorch Geometric and use *deeprobust.graph.data.Pyg2Dpr* to convert Pytorch Geometric data to DeepRobust. For example, we can first create an instance of the Dataset class and convert it to pytorch geometric data format.

```
from deeprobust.graph.data import Dataset, Dpr2Pyg, Pyg2Dpr
data = Dataset(root='/tmp/', name='cora') # load clean graph
pyg_data = Dpr2Pyg(data) # convert dpr to pyg
print(pyg_data)
```

(continues on next page)

(continued from previous page)

```
print(pyg_data[0])
dpr_data = Pyg2Dpr(pyg_data) # convert pyg to dpr
print(dpr_data.adj)
```

For the attacked graph `deeprobust.graph.PrePtbDataset`, it only has the attribute `adj`. To convert it to PyTorch Geometric data format, we can first convert the clean graph to Pyg and then update its `edge_index`:

```
from deeprobust.graph.data import Dataset, PrePtbDataset, Dpr2Pyg
data = Dataset(root='/tmp/', name='cora') # load clean graph
pyg_data = Dpr2Pyg(data) # convert dpr to pyg
# load perturbed graph
perturbed_data = PrePtbDataset(root='/tmp/',
        name='cora',
        attack_method='meta',
        ptb_rate=0.05)
perturbed_adj = perturbed_data.adj
pyg_data.update_edge_index(perturbed_adj) # inplace operation
```

Now `pyg_data` becomes the perturbed data in the format of PyTorch Geometric. We can then use it as the input for various Pytorch Geometric models!

5.2 Load OGB Datasets

Open Graph Benchmark (OGB) has provided various benchmark datasets. DeepRobsut now provides interface to convert OGB dataset format (Pyg data format) to DeepRobust format.

```
from ogb.nodeproppred import PygNodePropPredDataset
from deeprobust.graph.data import Pyg2Dpr
pyg_data = PygNodePropPredDataset(name = 'ogbn-arxiv')
dpr_data = Pyg2Dpr(pyg_data) # convert pyg to dpr
```

5.3 Load Pytorch Geometric Amazon and Coauthor Datasets

DeepRobust also provides access to the Amazon datasets and Coauthor datasets, i.e., *Amazon-Computers*, *Amazon-Photo*, *Coauthor-CS*, *Coauthor-Physics*, from Pytorch Geometric. Specifically, users can access them through *deeprobust.graph.data.AmazonPyg* and *deeprobust.graph.data.CoauthorPyg*. For example, we can directly load Amazon dataset from deeprobust in the format of pyg as follows,

```
from deeprobust.graph.data import AmazonPyg
computers = AmazonPyg(root='/tmp', name='computers')
print(computers)
print(computers[0])
photo = AmazonPyg(root='/tmp', name='photo')
print(photo)
print(photo[0])
```

Similarly, we can also load Coauthor dataset,

```
from deeprobust.graph.data import CoauthorPyg
cs = CoauthorPyg(root='/tmp', name='cs')
print(cs)
```

(continues on next page)

(continued from previous page)

```
print(cs[0])
physics = CoauthorPyg(root='/tmp', name='physics')
print(physics)
print(physics[0])
```

5.4 Working on PyTorch Geometric Models

In this subsection, we provide examples for using GNNs based on PyTorch Geometric. Spefically, we use GAT *deeprobust.graph.defense.GAT* and ChebNet *deeprobust.graph.defense.ChebNet* to further illustrate (while *deeprobust.graph.defense.SGC* is also available in this library). Basically, we can first convert the DeepRobust data to PyTorch Geometric data and then train Pyg models.

```
from deeprobust.graph.data import Dataset, Dpr2Pyg, PrePtbDataset
from deeprobust.graph.defense import GAT
data = Dataset(root='/tmp/', name='cora', seed=15)
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
gat = GAT(nfeat=features.shape[1],
          nhid=8, heads=8,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
gat = gat.to('cpu')
pyg_data = Dpr2Pyg(data) # convert deeprobust dataset to pyg dataset
gat.fit(pyg_data, patience=100, verbose=True) # train with earlystopping
gat.test() # test performance on clean graph

# load perturbed graph
perturbed_data = PrePtbDataset(root='/tmp/',
        name='cora',
        attack_method='meta',
        ptb_rate=0.05)
perturbed_adj = perturbed_data.adj
pyg_data.update_edge_index(perturbed_adj) # inplace operation
gat.fit(pyg_data, patience=100, verbose=True) # train with earlystopping
gat.test() # test performance on perturbed graph
```

```
from deeprobust.graph.data import Dataset, Dpr2Pyg
from deeprobust.graph.defense import ChebNet
data = Dataset(root='/tmp/', name='cora')
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
cheby = ChebNet(nfeat=features.shape[1],
          nhid=16, num_hops=3,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
cheby = cheby.to('cpu')
pyg_data = Dpr2Pyg(data) # convert deeprobust dataset to pyg dataset
cheby.fit(pyg_data, patience=10, verbose=True) # train with earlystopping
cheby.test()
```

5.5 More Details

More details can be found in test_gat.py, test_chebnet.py and test_sgc.py.

CHAPTER 6

Node Embedding Attack and Defense

In this section, we introduce the node embedding attack algorithms and corresponding victim models provided in DeepRobust.

6.1 Node Embedding Attack

Node embedding attack aims to fool node embedding models produce bad-quality embeddings. Specifically, DeepRobust provides the following node attack algorithms:

- *deeprobust.graph.global_attack.NodeEmbeddingAttack*
- *deeprobust.graph.global_attack.OtherNodeEmbeddingAttack*

They only take the adjacency matrix as input and the adjacency matrix is in the format of `scipy.sparse.csr_matrix`. You can specify the attack_type to either add edges or remove edges. Let's take a look at an example:

```
from deeprobust.graph.data import Dataset
from deeprobust.graph.global_attack import NodeEmbeddingAttack
data = Dataset(root='/tmp/', name='cora_ml', seed=15)
adj, features, labels = data.adj, data.features, data.labels
model = NodeEmbeddingAttack()
model.attack(adj, attack_type="remove")
modified_adj = model.modified_adj
model.attack(adj, attack_type="remove", min_span_tree=True)
modified_adj = model.modified_adj
model.attack(adj, attack_type="add", n_candidates=10000)
modified_adj = model.modified_adj
```

(continues on next page)

(continued from previous page)

```
model.attack(adj, attack_type="add_by_remove", n_candidates=10000)
modified_adj = model.modified_adj
```

The `OtherNodeEmbeddingAttack` contains the baseline methods reported in the paper Adversarial Attacks on Node Embeddings via Graph Poisoning. Aleksandar Bojchevski and Stephan Günnemann, ICML 2019. We can specify the type (chosen from *["degree", "eigencentrality", "random"]*) to generate corresponding attacks.

```
from deeprobust.graph.data import Dataset
from deeprobust.graph.global_attack import OtherNodeEmbeddingAttack
data = Dataset(root='/tmp/', name='cora_ml', seed=15)
adj, features, labels = data.adj, data.features, data.labels
model = OtherNodeEmbeddingAttack(type='degree')
model.attack(adj, attack_type="remove")
modified_adj = model.modified_adj
#
model = OtherNodeEmbeddingAttack(type='eigencentrality')
model.attack(adj, attack_type="remove")
modified_adj = model.modified_adj
#
model = OtherNodeEmbeddingAttack(type='random')
model.attack(adj, attack_type="add", n_candidates=10000)
modified_adj = model.modified_adj
```

6.2 Node Embedding Victim Models

DeepRobust provides two node embedding victim models, DeepWalk and Node2Vec:

- *deeprobust.graph.defense.DeepWalk*
- *deeprobust.graph.defense.Node2Vec*

There are three major functions in the two classes: `fit()`, `evaluate_node_classification()` and `evaluate_link_prediction`. The function `fit()` will train the node embdding models and store the embedding in `self.embedding`. For example,

```
from deeprobust.graph.data import Dataset
from deeprobust.graph.defense import DeepWalk
from deeprobust.graph.global_attack import NodeEmbeddingAttack
import numpy as np

dataset_str = 'cora_ml'
data = Dataset(root='/tmp/', name=dataset_str, seed=15)
adj, features, labels = data.adj, data.features, data.labels
idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test

print("Test DeepWalk on clean graph")
model = DeepWalk(type="skipgram")
model.fit(adj)
print(model.embedding)
```

After we trained the model, we can then test its performance on node classification and link prediction:

```
print("Test DeepWalk on node classification...")
# model.evaluate_node_classification(labels, idx_train, idx_test, lr_params={"max_iter
↪": 1000})
```

(continues on next page)

(continued from previous page)

```
model.evaluate_node_classification(labels, idx_train, idx_test)
print("Test DeepWalk on link prediciton...")
model.evaluate_link_prediction(adj, np.array(adj.nonzero()).T)
```

We can then test its performance on the attacked graph:

```
# set up the attack model
attacker = NodeEmbeddingAttack()
attacker.attack(adj, attack_type="remove", n_perturbations=1000)
modified_adj = attacker.modified_adj
print("Test DeepWalk on attacked graph")
model.fit(modified_adj)
model.evaluate_node_classification(labels, idx_train, idx_test)
```

CHAPTER 7

Image Attack and Defense

We introduce the usage of attacks and defense API in image package.

7.1 Attack Example

```
from deeprobust.image.attack.pgd import PGD
from deeprobust.image.config import attack_params
from deeprobust.image.utils import download_model
import torch
import deeprobust.image.netmodels.resnet as resnet

URL = "https://github.com/I-am-Bot/deeprobust_model/raw/master/CIFAR10_
↪ResNet18_epoch_50.pt"
download_model(URL, "$MODEL_PATH$")

model = resnet.ResNet18().to('cuda')
model.load_state_dict(torch.load("$MODEL_PATH$"))
model.eval()

transform_val = transforms.Compose([transforms.ToTensor()])
test_loader  = torch.utils.data.DataLoader(
             datasets.CIFAR10('deeprobust/image/data', train = False,␣
↪download=True,
             transform = transform_val),
             batch_size = 10, shuffle=True)

x, y = next(iter(test_loader))
```

(continues on next page)

(continued from previous page)

```
x = x.to('cuda').float()

adversary = PGD(model, device)
Adv_img = adversary.generate(x, y, **attack_params['PGD_CIFAR10'])
```

7.2 Defense Example

```
model = Net()
train_loader = torch.utils.data.DataLoader(
                datasets.MNIST('deeprobust/image/defense/data', train=True,␣
↪download=True,
                transform=transforms.Compose([transforms.ToTensor()])),
                batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(
               datasets.MNIST('deeprobust/image/defense/data', train=False,
               transform=transforms.Compose([transforms.ToTensor()])),
               batch_size=1000,shuffle=True)

defense = PGDtraining(model, 'cuda')
defense.generate(train_loader, test_loader, **defense_params["PGDtraining_
↪MNIST"])
```

CHAPTER 8

Package API

8.1 deeprobust.image.attack package

8.1.1 Submodules

8.1.2 deeprobust.image.attack.BPDA module

https://github.com/lordwarlock/Pytorch-BPDA/blob/master/bpda.py

8.1.3 deeprobust.image.attack.Nattack module

class NATTACK(*model*, *device='cuda'*)

Nattack is a black box attack algorithm.

generate(***kwargs*)

Call this function to generate adversarial examples.

Parameters **kwargs** – user defined paremeters

parse_params(*dataloader*, *classnum*, *target_or_not=False*, *clip_max=1*, *clip_min=0*, *epsilon=0.2*, *population=300*, *max_iterations=400*, *learning_rate=2*, *sigma=0.1*)

parse_params.

Parameters

- **dataloader** – dataloader
- **classnum** – classnum
- **target_or_not** – target_or_not
- **clip_max** – maximum pixel value
- **clip_min** – minimum pixel value
- **epsilon** – perturb constraint

- **population** – population
- **max_iterations** – maximum number of iterations
- **learning_rate** – learning rate
- **sigma** – sigma

8.1.4 deeprobust.image.attack.Universal module

https://github.com/ferjad/Universal_Adversarial_Perturbation_pytorch Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>

universal_adversarial_perturbation(*dataloader*, *model*, *device*, *xi=10*, *delta=0.2*, *max_iter_uni=10*, *p=inf*, *num_classes=10*, *overshoot=0.02*, *max_iter_df=10*, *t_p=0.2*)

universal_adversarial_perturbation.

Parameters

- **dataloader** – dataloader
- **model** – target model
- **device** – device
- **xi** – controls the l_p magnitude of the perturbation
- **delta** – controls the desired fooling rate (default = 80% fooling rate)
- **max_iter_uni** – maximum number of iteration (default = 10*num_images)
- **p** – norm to be used (default = np.inf)
- **num_classes** – num_classes (default = 10)
- **overshoot** – to prevent vanishing updates (default = 0.02)
- **max_iter_df** – maximum number of iterations for deepfool (default = 10)
- **t_p** – truth percentage, for how many flipped labels in a batch. (default = 0.2)

Returns

Return type the universal perturbation matrix.

8.1.5 deeprobust.image.attack.YOPOpgd module

class FASTPGD(*eps=0.023529411764705882*, *sigma=0.011764705882352941*, *nb_iter=20*, *norm=inf*, *DEVICE=<sphinx.ext.autodoc.importer._MockObject object>*, *mean=<sphinx.ext.autodoc.importer._MockObject object>*, *std=<sphinx.ext.autodoc.importer._MockObject object>*, *random_start=True*)

This module is the adversarial example gererated algorithm in YOPO.

References

Original code: https://github.com/a1600012888/YOPO-You-Only-Propagate-Once

single_attack(*net*, *inp*, *label*, *eta*, *target=None*)
Given the original image and the perturbation computed so far, computes a new perturbation. :param net: :param inp: original image :param label: :param eta: perturbation computed so far :return: a new perturbation

8.1.6 deeprobust.image.attack.base_attack module

class BaseAttack(*model*, *device='cuda'*)
Attack base class.

check_type_device(*image*, *label*)
Check device, match variable type to device type.

Parameters

- **image** – image
- **label** – label

generate(*image*, *label*, ***kwargs*)
Overide this function for the main body of attack algorithm.

Parameters

- **image** – original image
- **label** – original label
- **kwargs** – user defined parameters

parse_params(***kwargs*)
Parse user defined parameters.

8.1.7 deeprobust.image.attack.cw module

class CarliniWagner(*model*, *device='cuda'*)
C&W attack is an effective method to calcuate high-confidence adversarial examples.

References

This reimplementation is based on https://github.com/kkew3/pytorch-cw2 Copyright 2018 Kaiwen Wu

Examples

```
>>> from deeprobust.image.attack.cw import CarliniWagner
>>> from deeprobust.image.netmodels.CNN import Net
>>> from deeprobust.image.config import attack_params
```

```
>>> model = Net()
>>> model.load_state_dict(torch.load("./trained_models/MNIST_CNN_epoch_20.pt",␣
↪map_location = torch.device('cuda')))
>>> model.eval()
```

```
>>> x,y = datasets.MNIST()
>>> attack = CarliniWagner(model, device='cuda')
>>> AdvExArray = attack.generate(x, y, target_label = 1, classnum = 10, **attack_
↪params['CW_MNIST'])
```

generate (*image*, *label*, *target_label*, ***kwargs*)
Call this function to generate adversarial examples.

Parameters

- **image** – original image
- **label** – target label
- **kwargs** – user defined paremeters

loss_function (*x_p*, *const*, *target*, *reconstructed_original*, *confidence*, *min_*, *max_*)
Returns the loss and the gradient of the loss w.r.t. x, assuming that logits = model(x).

parse_params (*classnum=10*, *confidence=0.0001*, *clip_max=1*, *clip_min=0*, *max_iterations=1000*, *initial_const=0.01*, *binary_search_steps=5*, *learning_rate=1e-05*, *abort_early=True*)
Parse the user defined parameters.

Parameters

- **classnum** – number of class
- **confidence** – confidence
- **clip_max** – maximum pixel value
- **clip_min** – minimum pixel value
- **max_iterations** – maximum number of iterations
- **initial_const** – initialization of binary search
- **binary_search_steps** – step number of binary search
- **learning_rate** – learning rate
- **abort_early** – Set abort_early = True to allow early stop

pending_f (*x_p*)
Pending is the loss function is less than 0

to_model_space (*x*)
Transforms an input from the attack space to the model space. This transformation and the returned gradient are elementwise.

8.1.8 deeprobust.image.attack.deepfool module

class DeepFool (*model*, *device='cuda'*)
DeepFool attack.

generate (*image*, *label*, ***kwargs*)
Call this function to generate adversarial examples.

Parameters

- **image** (*1*H*W*3*) – original image
- **label** (*int*) – target label

- **kwargs** – user defined paremeters

Returns adversarial examples

Return type adv_img

parse_params(*num_classes=10*, *overshoot=0.02*, *max_iteration=50*)
Parse the user defined parameters

Parameters

- **num_classes** (*int*) – limits the number of classes to test against. (default = 10)
- **overshoot** (*float*) – used as a termination criterion to prevent vanishing updates (default = 0.02).
- **max_iteration** (*int*) – maximum number of iteration for deepfool (default = 50)

8.1.9 deeprobust.image.attack.fgsm module

class FGSM(*model*, *device='cuda'*)
FGSM attack is an one step gradient descent method.

generate(*image*, *label*, ***kwargs*)
" Call this function to generate FGSM adversarial examples.

Parameters

- **image** – original image
- **label** – target label
- **kwargs** – user defined paremeters

parse_params(*epsilon=0.2*, *order=inf*, *clip_max=None*, *clip_min=None*)
Parse the user defined parameters. :param model: victim model :param image: original attack images :param label: target labels :param epsilon: perturbation constraint :param order: constraint type :param clip_min: minimum pixel value :param clip_max: maximum pixel value :param device: device type, cpu or gpu

Returns perturbed images

Return type [N*C*H*W], floatTensor

8.1.10 deeprobust.image.attack.l2_attack module

8.1.11 deeprobust.image.attack.lbfgs module

class LBFGS(*model*, *device='cuda'*)
LBFGS is the first adversarial generating algorithm.

generate(*image*, *label*, *target_label*, ***kwargs*)
Call this function to generate adversarial examples.

Parameters

- **image** – original image
- **label** – target label
- **kwargs** – user defined paremeters

parse_params (*clip_max=1*, *clip_min=0*, *class_num=10*, *epsilon=1e-05*, *maxiter=20*)
Parse the user defined parameters.

Parameters

- **clip_max** – maximum pixel value
- **clip_min** – minimum pixel value
- **class_num** – total number of class
- **epsilon** – step length for binary seach
- **maxiter** – maximum number of iterations

8.1.12 deeprobust.image.attack.onepixel module

class Onepixel (*model*, *device='cuda'*)
Onepixel attack is an algorithm that allow attacker to only manipulate one (or a few) pixel to mislead classifier. This is a re-implementation of One pixel attack. Copyright (c) 2018 Debang Li

References

Akhtar, N., & Mian, A. (2018).Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey: A Survey. IEEE Access, 6, 14410-14430.

Reference code: https://github.com/DebangLi/one-pixel-attack-pytorch

generate (*image*, *label*, ***kwargs*)
Call this function to generate Onepixel adversarial examples.

Parameters

- **image** (*1 * 3 * W * H*) – original image
- **label** – target label
- **kwargs** – user defined paremeters

parse_params (*pixels=1*, *maxiter=100*, *popsize=400*, *samples=100*, *targeted_attack=False*, *print_log=True*, *target=0*)
Parse the user-defined params.

Parameters

- **pixels** – maximum number of manipulated pixels
- **maxiter** – maximum number of iteration
- **popsize** – population size
- **samples** – samples
- **targeted_attack** – targeted attack or not
- **print_log** – Set print_log = True to print out details in the searching algorithm
- **target** – target label (if targeted attack is set to be True)

8.1.13 deeprobust.image.attack.pgd module

class PGD(*model*, *device='cuda'*)
This is the multi-step version of FGSM attack.

generate(*image*, *label*, ***kwargs*)
Call this function to generate PGD adversarial examples.

Parameters

- **image** – original image
- **label** – target label
- **kwargs** – user defined paremeters

parse_params(*epsilon=0.03*, *num_steps=40*, *step_size=0.01*, *clip_max=1.0*, *clip_min=0.0*, *mean=(0, 0, 0)*, *std=(1.0, 1.0, 1, 0)*, *print_process=False*)
parse_params.

Parameters

- **epsilon** – perturbation constraint
- **num_steps** – iteration step
- **step_size** – step size
- **clip_max** – maximum pixel value
- **clip_min** – minimum pixel value
- **print_process** – whether to print out the log during optimization process, True or False print out the log during optimization process, True or False.

8.1.14 Module contents

8.2 deeprobust.image.defense package

8.2.1 Submodules

8.2.2 deeprobust.image.defense.LIDclassifier module

This is an implementation of LID detector. Currently this implementation is under testing.

References

Copyright (c) 2018 Xingjun Ma

get_lid(*model*, *X_test*, *X_test_noisy*, *X_test_adv*, *k*, *batch_size*)
get_lid.

Parameters

- **model** – model
- **X_test** – clean data
- **X_test_noisy** – noisy data
- **X_test_adv** – adversarial data

- **k** – k
- **batch_size** – batch_size

train (*self*, *device*, *train_loader*, *optimizer*, *epoch*)
train process.

Parameters

- **device** – device(option:'cpu', 'cuda')
- **train_loader** – train data loader
- **optimizer** – optimizer
- **epoch** – epoch

8.2.3 deeprobust.image.defense.TherEncoding module

This is an implementation of Thermometer Encoding.

References

Thermometer (*x*, *levels*, *flattened=False*)
Thermometer Encoding of the input.

one_hot (*x*, *levels*)
One hot Encoding of the input.

one_hot_to_thermometer (*x*, *levels*, *flattened=False*)
Convert One hot Encoding to Thermometer Encoding.

train (*model*, *device*, *train_loader*, *optimizer*, *epoch*)
training process.

Parameters

- **model** – model
- **device** – device
- **train_loader** – training data loader
- **optimizer** – optimizer
- **epoch** – epoch

8.2.4 deeprobust.image.defense.YOPO module

This is an implementation of adversarial training variant: YOPO. .. rubric:: References

You only propagate once: Painless adversarial training using maximal principle. arXiv preprint arXiv:1905.00877.

class CrossEntropyWithWeightPenlty (*module*, *DEVICE*, *reg_cof=0.0001*)

class Hamiltonian (*layer*, *reg_cof=0.0001*)

torch_accuracy (*output*, *target*, *topk=(1,)*) → List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f2786a68e90>]
param output, target: should be torch Variable

train_one_epoch(*net*, *batch_generator*, *optimizer*, *eps*, *criterion*, *LayerOneTrainner*, *K*, *DEVICE=<sphinx.ext.autodoc.importer._MockObject object>*, *descrip_str='Training'*)

Parameters

- **attack_freq** – Frequencies of training with adversarial examples. -1 indicates natural training
- **AttackMethod** – the attack method, None represents natural training

Returns None #(clean_acc, adv_acc)

8.2.5 deeprobust.image.defense.base_defense module

class BaseDefense(*model*, *device*)

Defense base class.

adv_data(*model*, *data*, *target*, ***kwargs*)

Generate adversarial examples for adversarial training. Overide this function to generate customize adv examples.

Parameters

- **model** – victim model
- **data** – original data
- **target** – target labels
- **kwargs** – parameters

loss(*output*, *target*)

Calculate training loss. Overide this function to customize loss.

Parameters

- **output** – model outputs
- **target** – true labels

parse_params(***kwargs*)

Parse user defined parameters

save_model()

Save model.

test(*test_loader*)

test.

Parameters **test_loader** – testing data

train(*train_loader*, *optimizer*, *epoch*)

train.

Parameters

- **train_loader** – training data
- **optimizer** – training optimizer
- **epoch** – training epoch

8.2.6 deeprobust.image.defense.fast module

This is an implementation of adversarial training variant: fast

References

class Fast(*model*, *device*)

adv_data(*data*, *output*, *ep=0.3*, *num_steps=40*)
Generate adversarial examples for adversarial training. Overide this function to generate customize adv examples.

Parameters

- **model** – victim model
- **data** – original data
- **target** – target labels
- **kwargs** – parameters

calculate_loss(*output*, *target*, *redmode='mean'*)
Calculate loss for training.

generate(*train_loader*, *test_loader*, ***kwargs*)
FGSM defense process:

parse_params(*save_dir='defense_models'*, *save_model=True*, *save_name='fast_mnist_fgsmtraining_0.2.pt'*, *epsilon=0.2*, *epoch_num=30*, *lr_train=0.005*, *momentum=0.1*)
Parse user defined parameters

test(*model*, *device*, *test_loader*)
Testing process.

train(*device*, *train_loader*, *optimizer*, *epoch*)
Training process.

8.2.7 deeprobust.image.defense.fgsmtraining module

This is the implementation of fgsm training.

References

..[1]Szegedy, C., Zaremba, W., Sutskever, I., Estrach, J. B., Erhan, D., Goodfellow, I., & Fergus, R. (2014, January). Intriguing properties of neural networks.

class FGSMtraining(*model*, *device*)
FGSM adversarial training.

adv_data(*data*, *output*, *ep=0.3*, *num_steps=40*)
Generate adversarial data for training.

Parameters

- **data** – data
- **output** – output
- **ep** – epsilon, perturbation budget.

- **num_steps** – iteration steps

calculate_loss (*output*, *target*, *redmode='mean'*)
Calculate loss for training.

generate (*train_loader*, *test_loader*, ***kwargs*)
FGSM adversarial training process.

Parameters

- **train_loader** – training data loader
- **test_loader** – testing data loader
- **kwargs** – kwargs

parse_params (*save_dir='defense_models'*, *save_model=True*, *save_name='mnist_fgsmtraining_0.2.pt'*, *epsilon=0.2*, *epoch_num=50*, *lr_train=0.005*, *momentum=0.1*)
parse_params.

Parameters

- **save_dir** – dir
- **save_model** – Whether to save model
- **save_name** – model name
- **epsilon** – attack perturbation constraint
- **epoch_num** – number of training epoch
- **lr_train** – training learning rate
- **momentum** – momentum for optimizor

test (*model*, *device*, *test_loader*)
testing process.

Parameters

- **model** – model
- **device** – device
- **test_loader** – testing dataloder

train (*device*, *train_loader*, *optimizer*, *epoch*)
training process.

Parameters

- **device** – device
- **train_loader** – training data loader
- **optimizer** – optimizer
- **epoch** – training epoch

8.2.8 deeprobust.image.defense.pgdtraining module

This is an implementation of pgd adversarial training. .. rubric:: References

..[1]Mądry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2017). Towards Deep Learning Models Resistant to Adversarial Attacks. stat, 1050, 9.

class PGDtraining(*model*, *device*)
PGD adversarial training.

adv_data(*data*, *output*, *ep=0.3*, *num_steps=10*, *perturb_step_size=0.01*)
Generate input(adversarial) data for training.

calculate_loss(*output*, *target*, *redmode='mean'*)
Calculate loss for training.

generate(*train_loader*, *test_loader*, ***kwargs*)
Call this function to generate robust model.

Parameters

- **train_loader** – training data loader
- **test_loader** – testing data loader
- **kwargs** – kwargs

parse_params(*epoch_num=100*, *save_dir='./defense_models'*, *save_name='mnist_pgdtraining_0.3'*, *save_model=True*, *epsilon=0.03137254901960784*, *num_steps=10*, *perturb_step_size=0.01*, *lr=0.1*, *momentum=0.1*, *save_per_epoch=10*)
Parameter parser.

Parameters

- **epoch_num** (*int*) – epoch
- **save_dir** (*str*) – model dir
- **save_name** (*str*) – model name
- **save_model** (*bool*) – Whether to save model
- **epsilon** (*float*) – attack constraint
- **num_steps** (*int*) – PGD attack iteration time
- **perturb_step_size** (*float*) – perturb step size
- **lr** (*float*) – learning rate for adversary training process
- **momentum** (*float*) – momentum for optimizor

test(*model*, *device*, *test_loader*)
testing process.

Parameters

- **model** – model
- **device** – device
- **test_loader** – testing dataloder

train(*device*, *train_loader*, *optimizer*, *epoch*)
training process.

Parameters

- **device** – device
- **train_loader** – training data loader
- **optimizer** – optimizer
- **epoch** – training epoch

8.2.9 deeprobust.image.defense.trades module

This is an implementation of [1] .. rubric:: References

Theoretically Principled Trade-off between Robustness and Accuracy. In International Conference on Machine Learning (pp. 7472-7482).

This implementation is based on their code: https://github.com/yaodongyu/TRADES Copyright (c) 2019 Hongyang Zhang, Yaodong Yu

class TRADES(*model*, *device='cuda'*)

TRADES.

generate(*train_loader*, *test_loader*, ***kwargs*)

generate robust model.

Parameters

- **train_loader** – train_loader
- **test_loader** – test_loader
- **kwargs** – kwargs

parse_params(*epochs=100*, *lr=0.01*, *momentum=0.9*, *epsilon=0.3*, *num_steps=40*, *step_size=0.01*, *beta=1.0*, *seed=1*, *log_interval=100*, *save_dir='./defense_model'*, *save_freq=10*)

:param epoch [int]

- pgd training epoch

:param save_dir [str]

- directory path to save model

:param epsilon [float]

- perturb constraint of pgd adversary example used to train defense model

:param num_steps [int]

- the perturb

:param perturb_step_size [float]

- step_size

:param lr [float]

- learning rate for adversary training process

:param momentum [float]

- parameter for optimizer in training process

test(*model*, *device*, *test_loader*)

test.

Parameters **test_loader** – testing data

train(*device*, *train_loader*, *optimizer*, *epoch*)

train.

Parameters

- **train_loader** – training data
- **optimizer** – training optimizer

- **epoch** – training epoch

8.2.10 Module contents

8.3 deeprobust.image.netmodels package

8.3.1 Submodules

8.3.2 deeprobust.image.netmodels.CNN module

This is an implementatio of a Convolution Neural Network with 2 Convolutional layer.

class Net (*in_channel1=1*, *out_channel1=32*, *out_channel2=64*, *H=28*, *W=28*)
Model counterparts.

test (*model*, *device*, *test_loader*)
test network.

Parameters

- **model** – model
- **device** – device(option:'cpu', 'cuda')
- **test_loader** – testing data loader

train (*model*, *device*, *train_loader*, *optimizer*, *epoch*)
train network.

Parameters

- **model** – model
- **device** – device(option:'cpu','cuda')
- **train_loader** – training data loader
- **optimizer** – optimizer
- **epoch** – epoch

8.3.3 deeprobust.image.netmodels.CNN_multilayer module

This is an implementation of Convolution Neural Network with multi conv layer.

class Net (*in_channel1=1*, *out_channel1=32*, *out_channel2=64*, *H=28*, *W=28*)

test (*model*, *device*, *test_loader*)
test.

Parameters

- **model** – model
- **device** – device
- **test_loader** – test_loader

train (*model*, *device*, *train_loader*, *optimizer*, *epoch*)
train.

Parameters

- **model** – model
- **device** – device
- **train_loader** – train_loader
- **optimizer** – optimizer
- **epoch** – epoch

8.3.4 deeprobust.image.netmodels.YOPOCNN module

Model for YOPO.

Reference

..[1]https://github.com/a1600012888/YOPO-You-Only-Propagate-Once

class Net(*drop=0.5*)

8.3.5 deeprobust.image.netmodels.densenet module

This is an implementation of DenseNet model.

Reference

..[1]Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. "Densely connected convolutional networks." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4700-4708. 2017.
..[2]Original implementation: https://github.com/kuangliu/pytorch-cifar

class Bottleneck(*in_planes*, *growth_rate*)

class DenseNet(*block*, *nblocks*, *growth_rate=12*, *reduction=0.5*, *num_classes=10*)
DenseNet.

DenseNet121()
DenseNet121.

DenseNet161()
DenseNet161.

DenseNet169()
DenseNet169.

DenseNet201()
DenseNet201.

class Transition(*in_planes*, *out_planes*)

densenet_cifar()
densenet_cifar.

test(*model*, *device*, *test_loader*)
test.

Parameters

- **model** – model
- **device** – device
- **test_loader** – test_loader

train (*model*, *device*, *train_loader*, *optimizer*, *epoch*)
train.

Parameters

- **model** – model
- **device** – device
- **train_loader** – train_loader
- **optimizer** – optimizer
- **epoch** – epoch

8.3.6 deeprobust.image.netmodels.preact_resnet module

This is an reimplementaiton of Pre-activation ResNet.

class PreActBlock (*in_planes*, *planes*, *stride=1*)
Pre-activation version of the BasicBlock.

class PreActBottleneck (*in_planes*, *planes*, *stride=1*)
Pre-activation version of the original Bottleneck module.

class PreActResNet (*block*, *num_blocks*, *num_classes=10*)
PreActResNet.

PreActResNet18 ()
PreActResNet18.

8.3.7 deeprobust.image.netmodels.resnet module

Properly implemented ResNet-s for CIFAR10 as described in paper [1].

This implementation is from Yerlan Idelbayev.

Reference

..[1] **Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun** Deep Residual Learning for Image Recognition. arXiv:1512.03385

..[2] https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

class BasicBlock (*in_planes*, *planes*, *stride=1*)

class Bottleneck (*in_planes*, *planes*, *stride=1*)

class Net (*block*, *num_blocks*, *num_classes=10*)

8.3.8 deeprobust.image.netmodels.train_model module

This function help to train model of different archtecture easily. Select model archtecture and training data, then output corresponding model.

train(*model*, *data*, *device*, *maxepoch*, *data_path='./'*, *save_per_epoch=10*, *seed=100*)
train.

Parameters

- **model** – model(option:'CNN', 'ResNet18', 'ResNet34', 'ResNet50', 'densenet', 'vgg11', 'vgg13', 'vgg16', 'vgg19')
- **data** – data(option:'MNIST','CIFAR10')
- **device** – device(option:'cpu', 'cuda')
- **maxepoch** – training epoch
- **data_path** – data path(default = './')
- **save_per_epoch** – save_per_epoch(default = 10)
- **seed** – seed

Examples

>>>import deeprobust.image.netmodels.train_model as trainmodel >>>trainmodel.train('CNN', 'MNIST', 'cuda', 20)

8.3.9 deeprobust.image.netmodels.train_resnet module

8.3.10 deeprobust.image.netmodels.vgg module

This is an implementation of VGG net.

Reference

..[1]Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014). ..[2]Original implementation: https://github.com/kuangliu/pytorch-cifar

class VGG(*vgg_name*)
VGG.

test(*model*, *device*, *test_loader*)
test.

Parameters

- **model** – model
- **device** – device
- **test_loader** – test_loader

train(*model*, *device*, *train_loader*, *optimizer*, *epoch*)
train.

Parameters

- **model** – model
- **device** – device
- **train_loader** – train_loader
- **optimizer** – optimizer
- **epoch** – epoch

8.3.11 Module contents

8.4 deeprobust.graph.global_attack package

8.4.1 Submodules

8.4.2 deeprobust.graph.global_attack.base_attack module

class BaseAttack(*model*, *nnodes*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
Abstract base class for target attack classes.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

attack(*ori_adj*, *n_perturbations*, ***kwargs*)
Generate attacks on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.

Returns

Return type None.

check_adj(*adj*)
Check if the modified adjacency is symmetric and unweighted.

check_adj_tensor(*adj*)
Check if the modified adjacency is symmetric, unweighted, all-zero diagonal.

save_adj(*root='/tmp/'*, *name='mod_adj'*)
Save attacked adjacency matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

save_features(*root='/tmp/'*, *name='mod_features'*)
Save attacked node feature matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

8.4.3 deeprobust.graph.global_attack.dice module

class DICE(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
As is described in ADVERSARIAL ATTACKS ON GRAPH NEURAL NETWORKS VIA META LEARNING (ICLR'19), 'DICE (delete internally, connect externally) is a baseline where, for each perturbation, we randomly choose whether to insert or remove an edge. Edges are only removed between nodes from the same classes, and only inserted between nodes from different classes.

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import DICE
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = DICE()
>>> model.attack(adj, labels, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_adj*, *labels*, *n_perturbations*, ***kwargs*)
Delete internally, connect externally. This baseline has all true class labels (train and test) available.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **labels** – node labels
- **n_perturbations** (*int*) – Number of edge removals/additions.

Returns

Return type None.

sample_forever(*adj*, *exclude*)
Randomly random sample edges from adjacency matrix, *exclude* is a set which contains the edges we do not want to sample and the ones already sampled

8.4.4 deeprobust.graph.global_attack.mettack module

Adversarial Attacks on Graph Neural Networks via Meta Learning. ICLR 2019 https://openreview.net/pdf?id=Bylnx209YX

Author Tensorflow implementation: https://github.com/danielzuegner/gnn-meta-attack

class BaseMeta(*model=None*, *nnodes=None*, *feature_shape=None*, *lambda_=0.5*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
Abstract base class for meta attack. Adversarial Attacks on Graph Neural Networks via Meta Learning, ICLR 2019, https://openreview.net/pdf?id=Bylnx209YX

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **lambda** (*float*) – **lambda_** is used to weight the two objectives in Eq. (10) in the paper.
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

attack(*adj*, *labels*, *n_perturbations*)
Generate attacks on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.

Returns

Return type None.

filter_potential_singletons(*modified_adj*)
Computes a mask for entries potentially leading to singleton nodes, i.e. one of the two nodes corresponding to the entry have degree 1 and there is an edge between the two nodes.

log_likelihood_constraint(*modified_adj*, *ori_adj*, *ll_cutoff*)
Computes a mask for entries that, if the edge corresponding to the entry is added/removed, would lead to the log likelihood constraint to be violated.

Note that different data type (float, double) can effect the final results.

class MetaApprox(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*, *with_bias=False*, *lambda_=0.5*, *train_iters=100*, *lr=0.01*)
Approximated version of Meta Attack. Adversarial Attacks on Graph Neural Networks via Meta Learning, ICLR 2019.

Examples

```
>>> import numpy as np
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import MetaApprox
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> model = MetaApprox(surrogate, nnodes=adj.shape[0], feature_shape=features.
↪shape,
        attack_structure=True, attack_features=False, device='cpu', lambda_=0).to(
↪'cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, idx_unlabeled, n_
↪perturbations=10, ll_constraint=True)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *idx_unlabeled*, *n_perturbations*, *ll_constraint=True*, *ll_cutoff=0.004*)

Generate n_perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_unlabeled** – unlabeled nodes indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **ll_constraint** (*bool*) – whether to exert the likelihood ratio test constraint
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95. It would be ignored if *ll_constraint* is False.

class Metattack(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*, *with_bias=False*, *lambda_=0.5*, *train_iters=100*, *lr=0.1*, *momentum=0.9*)

Meta attack. Adversarial Attacks on Graph Neural Networks via Meta Learning, ICLR 2019.

Examples

```
>>> import numpy as np
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import Metattack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> model = Metattack(surrogate, nnodes=adj.shape[0], feature_shape=features.
↪shape,
        attack_structure=True, attack_features=False, device='cpu', lambda_=0).to(
↪'cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, idx_unlabeled, n_
↪perturbations=10, ll_constraint=False)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *idx_unlabeled*, *n_perturbations*, *ll_constraint=True*, *ll_cutoff=0.004*)

Generate n_perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_unlabeled** – unlabeled nodes indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **ll_constraint** (*bool*) – whether to exert the likelihood ratio test constraint
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95. It would be ignored if *ll_constraint* is False.

8.4.5 deeprobust.graph.global_attack.nipa module

Non-target-specific Node Injection Attacks on Graph Neural Networks: A Hierarchical Reinforcement Learning Approach. WWW 2020. https://faculty.ist.psu.edu/vhonavar/Papers/www20.pdf

Still on testing stage. Haven't reproduced the performance yet.

class NIPA(*env*, *features*, *labels*, *idx_train*, *idx_val*, *idx_test*, *list_action_space*, *ratio*, *reward_type='binary'*, *batch_size=30*, *num_wrong=0*, *bilin_q=1*, *embed_dim=64*, *gm='mean_field'*, *mlp_hidden=64*, *max_lv=1*, *save_dir='checkpoint_dqn'*, *device=None*)

Reinforcement learning agent for NIPA attack. https://faculty.ist.psu.edu/vhonavar/Papers/www20.pdf

Parameters

- **env** – Node attack environment
- **features** – node features matrix
- **labels** – labels
- **idx_meta** – node meta indices
- **idx_test** – node test indices
- **list_action_space** (*list*) – list of action space
- **num_mod** – number of modification (perturbation) on the graph
- **reward_type** (*str*) – type of reward (e.g., 'binary')
- **batch_size** – batch size for training DQN
- **save_dir** – saving directory for model checkpoints
- **device** (*str*) – 'cpu' or 'cuda'

Examples

See more details in https://github.com/DSE-MSU/DeepRobust/blob/master/examples/graph/test_nipa.py

eval(*training=True*)

Evaluate RL agent.

possible_actions(*list_st*, *list_at*, *t*)

Parameters

- **list_st** – current state
- **list_at** – current action

Returns actions for next state

Return type list

train(*num_episodes=10*, *lr=0.01*)

Train RL agent.

8.4.6 deeprobust.graph.global_attack.random_attack module

class Random(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

Randomly adding edges to the input graph

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features

- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import Random
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = Random()
>>> model.attack(adj, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack (*ori_adj*, *n_perturbations*, *type='add'*, ***kwargs*)
Generate attacks on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.
- **type** (*str*) – perturbation type. Could be 'add', 'remove' or 'flip'.

Returns

Return type None.

inject_nodes (*adj*, *n_add*, *n_perturbations*)
For each added node, randomly connect with other nodes.

perturb_adj (*adj*, *n_perturbations*, *type='add'*)
Randomly add, remove or flip edges.

Parameters

- **adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.
- **type** (*str*) – perturbation type. Could be 'add', 'remove' or 'flip'.

Returns perturbed adjacency matrix

Return type scipy.sparse matrix

perturb_features (*features*, *n_perturbations*)
Randomly perturb features.

sample_forever (*adj*, *exclude*)
Randomly random sample edges from adjacency matrix, *exclude* is a set which contains the edges we do not want to sample and the ones already sampled

8.4.7 deeprobust.graph.global_attack.topology_attack module

Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective https://arxiv.org/pdf/1906.04214.pdf

Tensorflow Implementation: https://github.com/KaidiXu/GCN_ADV_Train

class MinMax(*model=None*, *nnodes=None*, *loss_type='CE'*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

MinMax attack for graph data.

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **loss_type** (*str*) – attack loss type, chosen from ['CE', 'CW']
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import MinMax
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Victim Model
>>> victim_model = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                       nhid=16, dropout=0.5, weight_decay=5e-4, device='cpu').to('cpu
↪')
>>> victim_model.fit(features, adj, labels, idx_train)
>>> # Setup Attack Model
>>> model = MinMax(model=victim_model, nnodes=adj.shape[0], loss_type='CE',␣
↪device='cpu').to('cpu')
>>> model.attack(features, adj, labels, idx_train, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *n_perturbations*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **epochs** – number of training epochs

class PGDAttack(*model=None*, *nnodes=None*, *loss_type='CE'*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

PGD attack for graph data.

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **loss_type** (*str*) – attack loss type, chosen from ['CE', 'CW']
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import PGDAttack
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Victim Model
>>> victim_model = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                  nhid=16, dropout=0.5, weight_decay=5e-4, device='cpu').to('cpu
↪')
>>> victim_model.fit(features, adj, labels, idx_train)
>>> # Setup Attack Model
>>> model = PGDAttack(model=victim_model, nnodes=adj.shape[0], loss_type='CE',␣
↪device='cpu').to('cpu')
>>> model.attack(features, adj, labels, idx_train, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *n_perturbations*, *epochs=200*, ***kwargs*)
Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **epochs** – number of training epochs

8.4.8 Module contents

class BaseAttack(*model*, *nnodes*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
Abstract base class for target attack classes.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

attack(*ori_adj*, *n_perturbations*, ***kwargs*)
Generate attacks on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.

Returns

Return type None.

check_adj(*adj*)
Check if the modified adjacency is symmetric and unweighted.

check_adj_tensor(*adj*)
Check if the modified adjacency is symmetric, unweighted, all-zero diagonal.

save_adj(*root='/tmp/'*, *name='mod_adj'*)
Save attacked adjacency matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

save_features(*root='/tmp/'*, *name='mod_features'*)
Save attacked node feature matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

class DICE(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
As is described in ADVERSARIAL ATTACKS ON GRAPH NEURAL NETWORKS VIA META LEARNING (ICLR'19), 'DICE (delete internally, connect externally) is a baseline where, for each perturbation, we randomly choose whether to insert or remove an edge. Edges are only removed between nodes from the same classes, and only inserted between nodes from different classes.

Parameters

- **model** – model to attack. Default *None*.

- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import DICE
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = DICE()
>>> model.attack(adj, labels, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_adj*, *labels*, *n_perturbations*, ***kwargs*)
Delete internally, connect externally. This baseline has all true class labels (train and test) available.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **labels** – node labels
- **n_perturbations** (*int*) – Number of edge removals/additions.

Returns

Return type None.

sample_forever(*adj*, *exclude*)
Randomly random sample edges from adjacency matrix, *exclude* is a set which contains the edges we do not want to sample and the ones already sampled

class MetaApprox(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*, *with_bias=False*, *lambda_=0.5*, *train_iters=100*, *lr=0.01*)
Approximated version of Meta Attack. Adversarial Attacks on Graph Neural Networks via Meta Learning, ICLR 2019.

Examples

```
>>> import numpy as np
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import MetaApprox
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                    nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
```

(continues on next page)

(continued from previous page)

```
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> model = MetaApprox(surrogate, nnodes=adj.shape[0], feature_shape=features.
↪shape,
        attack_structure=True, attack_features=False, device='cpu', lambda_=0).to(
↪'cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, idx_unlabeled, n_
↪perturbations=10, ll_constraint=True)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *idx_unlabeled*, *n_perturbations*, *ll_constraint=True*, *ll_cutoff=0.004*)

Generate n_perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_unlabeled** – unlabeled nodes indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **ll_constraint** (*bool*) – whether to exert the likelihood ratio test constraint
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95. It would be ignored if *ll_constraint* is False.

class Metattack(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*, *with_bias=False*, *lambda_=0.5*, *train_iters=100*, *lr=0.1*, *momentum=0.9*)

Meta attack. Adversarial Attacks on Graph Neural Networks via Meta Learning, ICLR 2019.

Examples

```
>>> import numpy as np
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import Metattack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> idx_unlabeled = np.union1d(idx_val, idx_test)
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
```

(continues on next page)

(continued from previous page)

```
>>> # Setup Attack Model
>>> model = Metattack(surrogate, nnodes=adj.shape[0], feature_shape=features.
↪shape,
        attack_structure=True, attack_features=False, device='cpu', lambda_=0).to(
↪'cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, idx_unlabeled, n_
↪perturbations=10, ll_constraint=False)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *idx_unlabeled*, *n_perturbations*, *ll_constraint=True*, *ll_cutoff=0.004*)

Generate n_perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_unlabeled** – unlabeled nodes indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **ll_constraint** (*bool*) – whether to exert the likelihood ratio test constraint
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95. It would be ignored if *ll_constraint* is False.

class Random(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

Randomly adding edges to the input graph

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import Random
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = Random()
>>> model.attack(adj, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_adj*, *n_perturbations*, *type='add'*, ***kwargs*)
Generate attacks on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.
- **type** (*str*) – perturbation type. Could be 'add', 'remove' or 'flip'.

Returns

Return type None.

inject_nodes(*adj*, *n_add*, *n_perturbations*)
For each added node, randomly connect with other nodes.

perturb_adj(*adj*, *n_perturbations*, *type='add'*)
Randomly add, remove or flip edges.

Parameters

- **adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of edge removals/additions.
- **type** (*str*) – perturbation type. Could be 'add', 'remove' or 'flip'.

Returns perturbed adjacency matrix

Return type scipy.sparse matrix

perturb_features(*features*, *n_perturbations*)
Randomly perturb features.

sample_forever(*adj*, *exclude*)
Randomly random sample edges from adjacency matrix, *exclude* is a set which contains the edges we do not want to sample and the ones already sampled

class MinMax(*model=None*, *nnodes=None*, *loss_type='CE'*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
MinMax attack for graph data.

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **loss_type** (*str*) – attack loss type, chosen from ['CE', 'CW']
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import MinMax
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Victim Model
>>> victim_model = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                       nhid=16, dropout=0.5, weight_decay=5e-4, device='cpu').to('cpu
↪')
>>> victim_model.fit(features, adj, labels, idx_train)
>>> # Setup Attack Model
>>> model = MinMax(model=victim_model, nnodes=adj.shape[0], loss_type='CE',␣
↪device='cpu').to('cpu')
>>> model.attack(features, adj, labels, idx_train, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *n_perturbations*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **epochs** – number of training epochs

class PGDAttack(*model=None*, *nnodes=None*, *loss_type='CE'*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

PGD attack for graph data.

Parameters

- **model** – model to attack. Default *None*.
- **nnodes** (*int*) – number of nodes in the input graph
- **loss_type** (*str*) – attack loss type, chosen from ['CE', 'CW']
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.global_attack import PGDAttack
>>> from deeprobust.graph.utils import preprocess
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> adj, features, labels = preprocess(adj, features, labels, preprocess_
↪adj=False) # conver to tensor
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Victim Model
>>> victim_model = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                       nhid=16, dropout=0.5, weight_decay=5e-4, device='cpu').to('cpu
↪')
>>> victim_model.fit(features, adj, labels, idx_train)
>>> # Setup Attack Model
>>> model = PGDAttack(model=victim_model, nnodes=adj.shape[0], loss_type='CE',␣
↪device='cpu').to('cpu')
>>> model.attack(features, adj, labels, idx_train, n_perturbations=10)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *n_perturbations*, *epochs=200*, ***kwargs*)
Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **epochs** – number of training epochs

class NIPA(*env*, *features*, *labels*, *idx_train*, *idx_val*, *idx_test*, *list_action_space*, *ratio*, *reward_type='binary'*, *batch_size=30*, *num_wrong=0*, *bilin_q=1*, *embed_dim=64*, *gm='mean_field'*, *mlp_hidden=64*, *max_lv=1*, *save_dir='checkpoint_dqn'*, *device=None*)
Reinforcement learning agent for NIPA attack. https://faculty.ist.psu.edu/vhonavar/Papers/www20.pdf

Parameters

- **env** – Node attack environment
- **features** – node features matrix
- **labels** – labels
- **idx_meta** – node meta indices
- **idx_test** – node test indices
- **list_action_space** (*list*) – list of action space
- **num_mod** – number of modification (perturbation) on the graph
- **reward_type** (*str*) – type of reward (e.g., 'binary')
- **batch_size** – batch size for training DQN
- **save_dir** – saving directory for model checkpoints

- **device** (*str*) – 'cpu' or 'cuda'

Examples

See more details in https://github.com/DSE-MSU/DeepRobust/blob/master/examples/graph/test_nipa.py

eval(*training=True*)
Evaluate RL agent.

possible_actions(*list_st*, *list_at*, *t*)

Parameters

- **list_st** – current state
- **list_at** – current action

Returns actions for next state

Return type list

train(*num_episodes=10*, *lr=0.01*)
Train RL agent.

class NodeEmbeddingAttack
Node embedding attack. Adversarial Attacks on Node Embeddings via Graph Poisoning. Aleksandar Bojchevski and Stephan Günnemann, ICML 2019 http://proceedings.mlr.press/v97/bojchevski19a.html

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import NodeEmbeddingAttack
>>> data = Dataset(root='/tmp/', name='cora_ml', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = NodeEmbeddingAttack()
>>> model.attack(adj, attack_type="remove")
>>> modified_adj = model.modified_adj
>>> model.attack(adj, attack_type="remove", min_span_tree=True)
>>> modified_adj = model.modified_adj
>>> model.attack(adj, attack_type="add", n_candidates=10000)
>>> modified_adj = model.modified_adj
>>> model.attack(adj, attack_type="add_by_remove", n_candidates=10000)
>>> modified_adj = model.modified_adj
```

attack(*adj*, *n_perturbations=1000*, *dim=32*, *window_size=5*, *attack_type='remove'*, *min_span_tree=False*, *n_candidates=None*, *seed=None*, ***kwargs*)
Selects the top (n_perturbations) number of flips using our perturbation attack.

Parameters

- **adj** – sp.spmatrix The graph represented as a sparse scipy matrix
- **n_perturbations** – int Number of flips to select
- **dim** – int Dimensionality of the embeddings.
- **window_size** – int Co-occurence window size.
- **attack_type** – str can be chosed from ["remove", "add", "add_by_remove"]
- **min_span_tree** – bool Whether to disallow edges that lie on the minimum spanning tree; only valid when *attack_type* is "remove"

- **n_candidates** – int Number of candiates for addition; only valid when *attack_type* is "add" or "add_by_remove";
- **seed** – int Random seed

flip_candidates (*adj*, *candidates*)

Flip the edges in the candidate set to non-edges and vise-versa.

Parameters

- **adj** – sp.csr_matrix, shape [n_nodes, n_nodes] Adjacency matrix of the graph
- **candidates** – np.ndarray, shape [?, 2] Candidate set of edge flips

Returns sp.csr_matrix, shape [n_nodes, n_nodes] Adjacency matrix of the graph with the flipped edges/non-edges.

generate_candidates_addition (*adj*, *n_candidates*, *seed=None*)

Generates candidate edge flips for addition (non-edge -> edge).

Parameters

- **adj** – sp.csr_matrix, shape [n_nodes, n_nodes] Adjacency matrix of the graph
- **n_candidates** – int Number of candidates to generate.
- **seed** – int Random seed

Returns np.ndarray, shape [?, 2] Candidate set of edge flips

generate_candidates_removal (*adj*, *seed=None*)

Generates candidate edge flips for removal (edge -> non-edge), disallowing one random edge per node to prevent singleton nodes.

Parameters

- **adj** – sp.csr_matrix, shape [n_nodes, n_nodes] Adjacency matrix of the graph
- **seed** – int Random seed

Returns np.ndarray, shape [?, 2] Candidate set of edge flips

generate_candidates_removal_minimum_spanning_tree (*adj*)

Generates candidate edge flips for removal (edge -> non-edge), disallowing edges that lie on the minimum spanning tree.

adj: sp.csr_matrix, shape [n_nodes, n_nodes] Adjacency matrix of the graph

Returns np.ndarray, shape [?, 2] Candidate set of edge flips

class OtherNodeEmbeddingAttack (*type*)

Baseline methods from the paper Adversarial Attacks on Node Embeddings via Graph Poisoning. Aleksandar Bojchevski and Stephan Günnemann, ICML 2019. http://proceedings.mlr.press/v97/bojchevski19a.html

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import OtherNodeEmbeddingAttack
>>> data = Dataset(root='/tmp/', name='cora_ml', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> model = OtherNodeEmbeddingAttack(type='degree')
>>> model.attack(adj, attack_type="remove")
```

(continues on next page)

(continued from previous page)

```
>>> modified_adj = model.modified_adj
>>> #
>>> model = OtherNodeEmbeddingAttack(type='eigencentrality')
>>> model.attack(adj, attack_type="remove")
>>> modified_adj = model.modified_adj
>>> #
>>> model = OtherNodeEmbeddingAttack(type='random')
>>> model.attack(adj, attack_type="add", n_candidates=10000)
>>> modified_adj = model.modified_adj
```

attack(*adj*, *n_perturbations=1000*, *attack_type='remove'*, *min_span_tree=False*, *n_candidates=None*, *seed=None*, ***kwargs*)

Selects the top (n_perturbations) number of flips using our perturbation attack.

Parameters

- **adj** – sp.spmatrix The graph represented as a sparse scipy matrix
- **n_perturbations** – int Number of flips to select
- **dim** – int Dimensionality of the embeddings.
- **attack_type** – str can be chosed from ["remove", "add"]
- **min_span_tree** – bool Whether to disallow edges that lie on the minimum spanning tree; only valid when *attack_type* is "remove"
- **n_candidates** – int Number of candiates for addition; only valid when *attack_type* is "add";
- **seed** – int Random seed;

Returns np.ndarray, shape [?, 2] The top edge flips from the candidate set

degree_top_flips(*adj*, *candidates*, *n_perturbations*, *complement*)

Selects the top (n_perturbations) number of flips using degree centrality score of the edges.

Parameters

- **adj** – sp.spmatrix The graph represented as a sparse scipy matrix
- **candidates** – np.ndarray, shape [?, 2] Candidate set of edge flips
- **n_perturbations** – int Number of flips to select
- **complement** – bool Whether to look at the complement graph

Returns np.ndarray, shape [?, 2] The top edge flips from the candidate set

eigencentrality_top_flips(*adj*, *candidates*, *n_perturbations*)

Selects the top (n_perturbations) number of flips using eigencentrality score of the edges. Applicable only when removing edges.

Parameters

- **adj** – sp.spmatrix The graph represented as a sparse scipy matrix
- **candidates** – np.ndarray, shape [?, 2] Candidate set of edge flips
- **n_perturbations** – int Number of flips to select

Returns np.ndarray, shape [?, 2] The top edge flips from the candidate set

random_top_flips(*candidates*, *n_perturbations*, *seed=None*)

Selects (n_perturbations) number of flips at random.

Parameters

- **candidates** – np.ndarray, shape [?, 2] Candidate set of edge flips
- **n_perturbations** – int Number of flips to select
- **seed** – int Random seed

Returns np.ndarray, shape [?, 2] The top edge flips from the candidate set

8.5 deeprobust.graph.targeted_attack package

8.5.1 Submodules

8.5.2 deeprobust.graph.targeted_attack.base_attack module

class BaseAttack(*model*, *nnodes*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
Abstract base class for target attack classes.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

attack(*ori_adj*, *n_perturbations*, ***kwargs*)
Generate perturbations on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

Returns

Return type None.

check_adj(*adj*)
Check if the modified adjacency is symmetric and unweighted.

save_adj(*root='/tmp/'*, *name='mod_adj'*)
Save attacked adjacency matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

save_features(*root='/tmp/'*, *name='mod_features'*)
Save attacked node feature matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

8.5.3 deeprobust.graph.targeted_attack.fga module

FGA: Fast Gradient Attack on Network Embedding (https://arxiv.org/pdf/1809.02797.pdf) Another very similar algorithm to mention here is FGSM (for graph data). It is mentioned in Zugner's paper, Adversarial Attacks on Neural Networks for Graph Data, KDD'19

class FGA(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

FGA/FGSM.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import FGA
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> target_node = 0
>>> model = FGA(surrogate, nnodes=adj.shape[0], attack_structure=True, attack_
↪features=False, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, *verbose=False*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix
- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) node feature matrix
- **labels** – node labels
- **idx_train** – training node indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

8.5.4 deeprobust.graph.targeted_attack.ig_attack module

Adversarial Examples on Graph Data: Deep Insights into Attack and Defense https://arxiv.org/pdf/1903.01610.pdf

class IGAttack(*model*, *nnodes=None*, *feature_shape=None*, *attack_structure=True*, *attack_features=True*, *device='cpu'*)

IGAttack: IG-FGSM. Adversarial Examples on Graph Data: Deep Insights into Attack and Defense, https://arxiv.org/pdf/1903.01610.pdf.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import IGAttack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> target_node = 0
>>> model = IGAttack(surrogate, nnodes=adj.shape[0], attack_structure=True,␣
↪attack_features=True, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, target_node, n_perturbations=5,
↪ steps=10)
```

(continues on next page)

(continued from previous page)

```
>>> modified_adj = model.modified_adj
>>> modified_features = model.modified_features
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, *steps=10*, ***kwargs*)
Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – training nodes indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **steps** (*int*) – steps for computing integrated gradients

calc_importance_edge(*features*, *adj_norm*, *labels*, *steps*)
Calculate integrated gradient for edges. Although I think the the gradient should be with respect to adj instead of adj_norm, but the calculation is too time-consuming. So I finally decided to calculate the gradient of loss with respect to adj_norm

calc_importance_feature(*features*, *adj_norm*, *labels*, *steps*)
Calculate integrated gradient for features

8.5.5 deeprobust.graph.targeted_attack.nettack module

Adversarial Attacks on Neural Networks for Graph Data. KDD 2018. https://arxiv.org/pdf/1805.07984.pdf

Author's Implementation https://github.com/danielzuegner/nettack

Since pytorch does not have good enough support to the operations on sparse tensor, this part of code is heavily based on the author's implementation.

class Nettack(*model*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
Nettack.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import Nettack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
              nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> target_node = 0
>>> model = Nettack(surrogate, nnodes=adj.shape[0], attack_structure=True, attack_
↪features=True, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
>>> modified_features = model.modified_features
```

attack(*features*, *adj*, *labels*, *target_node*, *n_perturbations*, *direct=True*, *n_influencers=0*, *ll_cutoff=0.004*, *verbose=True*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** (*torch.Tensor or scipy.sparse.csr_matrix*) – Origina (unperturbed) node feature matrix. Note that torch.Tensor will be automatically transformed into scipy.sparse.csr_matrix
- **ori_adj** (*torch.Tensor or scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix. Note that torch.Tensor will be automatically transformed into scipy.sparse.csr_matrix
- **labels** – node labels
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **direct** (*bool*) – whether to conduct direct attack
- **n_influencers** – number of influencer nodes when performing indirect attack. (setting *direct* to False). When *direct* is True, it would be ignored.
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95.
- **verbose** (*bool*) – whether to show verbose logs

compute_cooccurrence_constraint(*nodes*)

Co-occurrence constraint as described in the paper.

Parameters **nodes** (*np.array*) – Nodes whose features are considered for change

Returns Binary matrix of dimension len(nodes) x D. A 1 in entry n,d indicates that we are allowed to add feature d to the features of node n.

Return type np.array [len(nodes), D], dtype bool

compute_new_a_hat_uv(*potential_edges*, *target_node*)
Compute the updated A_hat_square_uv entries that would result from inserting/deleting the input edges, for every edge.

Parameters **potential_edges** (*np.array, shape [P,2], dtype int*) – The edges to check.

Returns sp.sparse_matrix

Return type updated A_hat_square_u entries, a sparse PxN matrix, where P is len(possible_edges)

feature_scores()
Compute feature scores for all possible feature changes.

filter_potential_singletons(*modified_adj*)
Computes a mask for entries potentially leading to singleton nodes, i.e. one of the two nodes corresponding to the entry have degree 1 and there is an edge between the two nodes.

get_attacker_nodes(*n=5*, *add_additional_nodes=False*)
Determine the influencer nodes to attack node i based on the weights W and the attributes X.

reset()
Reset Nettack

struct_score(*a_hat_uv*, *XW*)
Compute structure scores, cf. Eq. 15 in the paper

Parameters

- **a_hat_uv** (*sp.sparse_matrix, shape [P,2]*) – Entries of matrix A_hat^2_u for each potential edge (see paper for explanation)
- **XW** (*sp.sparse_matrix, shape [N, K], dtype float*) – The class logits for each node.

Returns The struct score for every row in a_hat_uv

Return type np.array [P,]

compute_alpha(*n*, *S_d*, *d_min*)
Approximate the alpha of a power law distribution.

compute_log_likelihood(*n*, *alpha*, *S_d*, *d_min*)
Compute log likelihood of the powerlaw fit.

compute_new_a_hat_uv
Compute the new values [A_hat_square]_u for every potential edge, where u is the target node. C.f. Theorem 5.1 equation 17.

filter_singletons(*edges*, *adj*)
Filter edges that, if removed, would turn one or more nodes into singleton nodes.

update_Sx(*S_old*, *n_old*, *d_old*, *d_new*, *d_min*)
Update on the sum of log degrees S_d and n based on degree distribution resulting from inserting or deleting a single edge.

8.5.6 deeprobust.graph.targeted_attack.rl_s2v module

Adversarial Attacks on Neural Networks for Graph Data. ICML 2018. https://arxiv.org/abs/1806.02371

Author's Implementation https://github.com/Hanjun-Dai/graph_adversarial_attack

This part of code is adopted from the author's implementation (Copyright (c) 2018 Dai, Hanjun and Li, Hui and Tian, Tian and Huang, Xin and Wang, Lin and Zhu, Jun and Song, Le) but modified to be integrated into the repository.

class RLS2V(*env*, *features*, *labels*, *idx_meta*, *idx_test*, *list_action_space*, *num_mod*, *reward_type*, *batch_size=10*, *num_wrong=0*, *bilin_q=1*, *embed_dim=64*, *gm='mean_field'*, *mlp_hidden=64*, *max_lv=1*, *save_dir='checkpoint_dqn'*, *device=None*)

Reinforcement learning agent for RL-S2V attack.

Parameters

- **env** – Node attack environment
- **features** – node features matrix
- **labels** – labels
- **idx_meta** – node meta indices
- **idx_test** – node test indices
- **list_action_space** (*list*) – list of action space
- **num_mod** – number of modification (perturbation) on the graph
- **reward_type** (*str*) – type of reward (e.g., 'binary')
- **batch_size** – batch size for training DQN
- **save_dir** – saving directory for model checkpoints
- **device** (*str*) – 'cpu' or 'cuda'

Examples

See details in https://github.com/DSE-MSU/DeepRobust/blob/master/examples/graph/test_rl_s2v.py

eval(*training=True*)

Evaluate RL agent.

train(*num_steps=100000*, *lr=0.001*)

Train RL agent.

8.5.7 deeprobust.graph.targeted_attack.rnd module

class RND(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

As is described in Adversarial Attacks on Neural Networks for Graph Data (KDD'19), 'Rnd is an attack in which we modify the structure of the graph. Given our target node v, in each step we randomly sample nodes u whose lable is different from v and add the edge u,v to the graph structure

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.targeted_attack import RND
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Attack Model
>>> target_node = 0
>>> model = RND()
>>> # Attack
>>> model.attack(adj, labels, idx_train, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
>>> # # You can also inject nodes
>>> # model.add_nodes(features, adj, labels, idx_train, target_node, n_added=10,␣
↪n_perturbations=100)
>>> # modified_adj = model.modified_adj
```

add_nodes(*features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_added=1*, *n_perturbations=10*, ***kwargs*)

For each added node, first connect the target node with added fake nodes. Then randomly connect the fake nodes with other nodes whose label is different from target node. As for the node feature, simply copy arbitary node

attack(*ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, ***kwargs*)

Randomly sample nodes u whose lable is different from v and add the edge u,v to the graph structure. This baseline only has access to true class labels in training set

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

8.5.8 Module contents

class BaseAttack(*model*, *nnodes*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

Abstract base class for target attack classes.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

attack(*ori_adj*, *n_perturbations*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix.
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

Returns

Return type None.

check_adj(*adj*)
Check if the modified adjacency is symmetric and unweighted.

save_adj(*root='/tmp/'*, *name='mod_adj'*)
Save attacked adjacency matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

save_features(*root='/tmp/'*, *name='mod_features'*)
Save attacked node feature matrix.

Parameters

- **root** – root directory where the variable should be saved
- **name** (*str*) – saved file name

Returns

Return type None.

class FGA(*model*, *nnodes*, *feature_shape=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)
FGA/FGSM.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import FGA
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
```

(continues on next page)

(continued from previous page)

```
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> target_node = 0
>>> model = FGA(surrogate, nnodes=adj.shape[0], attack_structure=True, attack_
↪features=False, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, *verbose=False*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix
- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) node feature matrix
- **labels** – node labels
- **idx_train** – training node indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

class RND(*model=None*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

As is described in Adversarial Attacks on Neural Networks for Graph Data (KDD'19), 'Rnd is an attack in which we modify the structure of the graph. Given our target node v, in each step we randomly sample nodes u whose lable is different from v and add the edge u,v to the graph structure

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.targeted_attack import RND
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Attack Model
>>> target_node = 0
```

(continues on next page)

(continued from previous page)

```
>>> model = RND()
>>> # Attack
>>> model.attack(adj, labels, idx_train, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
>>> # # You can also inject nodes
>>> # model.add_nodes(features, adj, labels, idx_train, target_node, n_added=10,␣
↪n_perturbations=100)
>>> # modified_adj = model.modified_adj
```

add_nodes(*features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_added=1*, *n_perturbations=10*, ***kwargs*)

For each added node, first connect the target node with added fake nodes. Then randomly connect the fake nodes with other nodes whose label is different from target node. As for the node feature, simply copy arbitary node

attack(*ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, ***kwargs*)

Randomly sample nodes u whose lable is different from v and add the edge u,v to the graph structure. This baseline only has access to true class labels in training set

Parameters

- **ori_adj** (*scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – node training indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.

class Nettack(*model*, *nnodes=None*, *attack_structure=True*, *attack_features=False*, *device='cpu'*)

Nettack.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – ‘cpu’ or ‘cuda’

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import Nettack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                    nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
```

(continues on next page)

(continued from previous page)

```
>>> # Setup Attack Model
>>> target_node = 0
>>> model = Nettack(surrogate, nnodes=adj.shape[0], attack_structure=True, attack_
↪features=True, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, target_node, n_perturbations=5)
>>> modified_adj = model.modified_adj
>>> modified_features = model.modified_features
```

attack(*features*, *adj*, *labels*, *target_node*, *n_perturbations*, *direct=True*, *n_influencers=0*, *ll_cutoff=0.004*, *verbose=True*, ***kwargs*)

Generate perturbations on the input graph.

Parameters

- **ori_features** (*torch.Tensor or scipy.sparse.csr_matrix*) – Origina (unperturbed) node feature matrix. Note that torch.Tensor will be automatically transformed into scipy.sparse.csr_matrix
- **ori_adj** (*torch.Tensor or scipy.sparse.csr_matrix*) – Original (unperturbed) adjacency matrix. Note that torch.Tensor will be automatically transformed into scipy.sparse.csr_matrix
- **labels** – node labels
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **direct** (*bool*) – whether to conduct direct attack
- **n_influencers** – number of influencer nodes when performing indirect attack. (setting *direct* to False). When *direct* is True, it would be ignored.
- **ll_cutoff** (*float*) – The critical value for the likelihood ratio test of the power law distributions. See the Chi square distribution with one degree of freedom. Default value 0.004 corresponds to a p-value of roughly 0.95.
- **verbose** (*bool*) – whether to show verbose logs

compute_cooccurrence_constraint(*nodes*)

Co-occurrence constraint as described in the paper.

Parameters **nodes** (*np.array*) – Nodes whose features are considered for change

Returns Binary matrix of dimension len(nodes) x D. A 1 in entry n,d indicates that we are allowed to add feature d to the features of node n.

Return type np.array [len(nodes), D], dtype bool

compute_new_a_hat_uv(*potential_edges*, *target_node*)

Compute the updated A_hat_square_uv entries that would result from inserting/deleting the input edges, for every edge.

Parameters **potential_edges** (*np.array, shape [P,2], dtype int*) – The edges to check.

Returns sp.sparse_matrix

Return type updated A_hat_square_u entries, a sparse PxN matrix, where P is len(possible_edges)

feature_scores()
Compute feature scores for all possible feature changes.

filter_potential_singletons(*modified_adj*)
Computes a mask for entries potentially leading to singleton nodes, i.e. one of the two nodes corresponding to the entry have degree 1 and there is an edge between the two nodes.

get_attacker_nodes(*n=5*, *add_additional_nodes=False*)
Determine the influencer nodes to attack node i based on the weights W and the attributes X.

reset()
Reset Nettack

struct_score(*a_hat_uv*, *XW*)
Compute structure scores, cf. Eq. 15 in the paper

Parameters

- **a_hat_uv** (*sp.sparse_matrix, shape [P,2]*) – Entries of matrix A_hat^2_u for each potential edge (see paper for explanation)
- **XW** (*sp.sparse_matrix, shape [N, K], dtype float*) – The class logits for each node.

Returns The struct score for every row in a_hat_uv

Return type np.array [P,]

class IGAttack(*model*, *nnodes=None*, *feature_shape=None*, *attack_structure=True*, *attack_features=True*, *device='cpu'*)
IGAttack: IG-FGSM. Adversarial Examples on Graph Data: Deep Insights into Attack and Defense, https://arxiv.org/pdf/1903.01610.pdf.

Parameters

- **model** – model to attack
- **nnodes** (*int*) – number of nodes in the input graph
- **feature_shape** (*tuple*) – shape of the input node features
- **attack_structure** (*bool*) – whether to attack graph structure
- **attack_features** (*bool*) – whether to attack node features
- **device** (*str*) – 'cpu' or 'cuda'

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> from deeprobust.graph.targeted_attack import IGAttack
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # Setup Surrogate model
>>> surrogate = GCN(nfeat=features.shape[1], nclass=labels.max().item()+1,
                    nhid=16, dropout=0, with_relu=False, with_bias=False, device='cpu
↪').to('cpu')
>>> surrogate.fit(features, adj, labels, idx_train, idx_val, patience=30)
>>> # Setup Attack Model
>>> target_node = 0
```

(continues on next page)

(continued from previous page)

```
>>> model = IGAttack(surrogate, nnodes=adj.shape[0], attack_structure=True,␣
↪attack_features=True, device='cpu').to('cpu')
>>> # Attack
>>> model.attack(features, adj, labels, idx_train, target_node, n_perturbations=5,
↪ steps=10)
>>> modified_adj = model.modified_adj
>>> modified_features = model.modified_features
```

attack(*ori_features*, *ori_adj*, *labels*, *idx_train*, *target_node*, *n_perturbations*, *steps=10*, ***kwargs*)
Generate perturbations on the input graph.

Parameters

- **ori_features** – Original (unperturbed) node feature matrix
- **ori_adj** – Original (unperturbed) adjacency matrix
- **labels** – node labels
- **idx_train** – training nodes indices
- **target_node** (*int*) – target node index to be attacked
- **n_perturbations** (*int*) – Number of perturbations on the input graph. Perturbations could be edge removals/additions or feature removals/additions.
- **steps** (*int*) – steps for computing integrated gradients

calc_importance_edge(*features*, *adj_norm*, *labels*, *steps*)
Calculate integrated gradient for edges. Although I think the the gradient should be with respect to adj instead of adj_norm, but the calculation is too time-consuming. So I finally decided to calculate the gradient of loss with respect to adj_norm

calc_importance_feature(*features*, *adj_norm*, *labels*, *steps*)
Calculate integrated gradient for features

class RLS2V(*env*, *features*, *labels*, *idx_meta*, *idx_test*, *list_action_space*, *num_mod*, *reward_type*, *batch_size=10*, *num_wrong=0*, *bilin_q=1*, *embed_dim=64*, *gm='mean_field'*, *mlp_hidden=64*, *max_lv=1*, *save_dir='checkpoint_dqn'*, *device=None*)
Reinforcement learning agent for RL-S2V attack.

Parameters

- **env** – Node attack environment
- **features** – node features matrix
- **labels** – labels
- **idx_meta** – node meta indices
- **idx_test** – node test indices
- **list_action_space** (*list*) – list of action space
- **num_mod** – number of modification (perturbation) on the graph
- **reward_type** (*str*) – type of reward (e.g., 'binary')
- **batch_size** – batch size for training DQN
- **save_dir** – saving directory for model checkpoints
- **device** (*str*) – 'cpu' or 'cuda'

Examples

See details in https://github.com/DSE-MSU/DeepRobust/blob/master/examples/graph/test_rl_s2v.py

eval(*training=True*)
Evaluate RL agent.

train(*num_steps=100000*, *lr=0.001*)
Train RL agent.

8.6 deeprobust.graph.defense package

8.6.1 Submodules

8.6.2 deeprobust.graph.defense.adv_training module

class AdvTraining(*model*, *adversary=None*, *device='cpu'*)
Adversarial training framework for defending against attacks.

Parameters

- **model** – model to protect, e.g, GCN
- **adversary** – attack model
- **device** (*str*) – 'cpu' or 'cuda'

adv_train(*features*, *adj*, *labels*, *idx_train*, *train_iters*, ***kwargs*)
Start adversarial training.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **train_iters** (*int*) – number of training epochs

8.6.3 deeprobust.graph.defense.gcn module

class GCN(*nfeat*, *nhid*, *nclass*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device=None*)
2 Layer Graph Convolutional Network.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN

- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GCN.

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> gcn = GCN(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
>>> gcn = gcn.to('cpu')
>>> gcn.fit(features, adj, labels, idx_train) # train without earlystopping
>>> gcn.fit(features, adj, labels, idx_train, idx_val, patience=30) # train with␣
↪earlystopping
>>> gcn.test(idx_test)
```

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *train_iters=200*, *initialize=True*, *verbose=False*, *normalize=True*, *patience=500*, ***kwargs*)
Train the gcn model, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs
- **normalize** (*bool*) – whether to normalize the input adjacency matrix.
- **patience** (*int*) – patience for early stopping, only valid when *idx_val* is given

initialize()
Initialize parameters of GCN.

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCN

Return type torch.FloatTensor

test(*idx_test*)
Evaluate GCN performance on test set.

Parameters **idx_test** – node testing indices

class GraphConvolution(*in_features*, *out_features*, *with_bias=True*)
Simple GCN layer, similar to https://github.com/tkipf/pygcn

forward(*input*, *adj*)
Graph Convolutional Layer forward function

8.6.4 deeprobust.graph.defense.gcn_preprocess module

class GCNJaccard(*nfeat*, *nhid*, *nclass*, *binary_feature=True*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device='cpu'*)
GCNJaccard first preprocesses input graph via droppining dissimilar edges and train a GCN based on the processed graph. See more details in Adversarial Examples on Graph Data: Deep Insights into Attack and Defense, https://arxiv.org/pdf/1903.01610.pdf.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GCNJaccard.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import GCNJaccard
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
```

(continues on next page)

(continued from previous page)

```
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = GCNJaccard(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu').to('cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val, threshold=0.03)
```

drop_dissimilar_edges (*features*, *adj*, *metric='similarity'*)
Drop dissimilar edges.(Faster version using numba)

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *threshold=0.01*, *train_iters=200*, *initialize=True*, *verbose=True*, ***kwargs*)
First drop dissimilar edges with similarity smaller than given threshold and then train the gcn model on the processed graph. When idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features. The format can be numpy.array or scipy matrix
- **adj** – the adjacency matrix.
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **threshold** (*float*) – similarity threshold for dropping edges. If two connected nodes with similarity smaller than threshold, the edge between them will be removed.
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCNJaccard

Return type torch.FloatTensor

class GCNSVD (*nfeat*, *nhid*, *nclass*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device='cpu'*)
GCNSVD is a 2 Layer Graph Convolutional Network with Truncated SVD as preprocessing. See more details in All You Need Is Low (Rank): Defending Against Adversarial Attacks on Graphs, https://dl.acm.org/doi/abs/10.1145/3336191.3371789.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GCNSVD.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import GCNSVD
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = GCNSVD(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu').to('cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val, k=20)
```

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *k=50*, *train_iters=200*, *initialize=True*, *verbose=True*, ***kwargs*)
First perform rank-k approximation of adjacency matrix via truncated SVD, and then train the gcn model on the processed graph, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **k** (*int*) – number of singular values and vectors to compute.
- **train_iters** (*int*) – number of training epochs

- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCNSVD

Return type torch.FloatTensor

truncatedSVD (*data*, *k=50*)
Truncated SVD on input data.

Parameters

- **data** – input matrix to be decomposed
- **k** (*int*) – number of singular values and vectors to compute.

Returns reconstructed matrix.

Return type numpy.array

8.6.5 deeprobust.graph.defense.pgd module

class PGD (*params*, *proxs*, *alphas*, *lr=<sphinx.ext.autodoc.importer._MockObject object>*, *momentum=0*, *dampening=0*, *weight_decay=0*)
Proximal gradient descent.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **proxs** (*iterable*) – iterable of proximal operators
- **alpha** (*iterable*) – iterable of coefficients for proximal gradient descent
- **lr** (*float*) – learning rate
- **momentum** (*float*) – momentum factor (default: 0)
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float*) – dampening for momentum (default: 0)

class ProxOperators
Proximal Operators.

prox_l1 (*data*, *alpha*)
Proximal operator for l1 norm.

prox_nuclear (*data*, *alpha*)
Proximal operator for nuclear norm (trace norm).

class SGD(*params*, *lr=<sphinx.ext.autodoc.importer._MockObject object>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*)

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

8.6.6 deeprobust.graph.defense.prognn module

class EstimateAdj(*adj*, *symmetric=False*, *device='cpu'*)
Provide a pytorch parameter matrix for estimated adjacency matrix and corresponding operations.

class ProGNN(*model*, *args*, *device*)
ProGNN (Properties Graph Neural Network). See more details in Graph Structure Learning for Robust Graph Neural Networks, KDD 2020, https://arxiv.org/abs/2005.10203.

Parameters

- **model** – model: The backbone GNN model in ProGNN
- **args** – model configs
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

See details in https://github.com/ChandlerBang/Pro-GNN.

fit(*features*, *adj*, *labels*, *idx_train*, *idx_val*, ***kwargs*)
Train Pro-GNN.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices

test(*features*, *labels*, *idx_test*)
Evaluate the performance of ProGNN on test set

8.6.7 deeprobust.graph.defense.r_gcn module

Robust Graph Convolutional Networks Against Adversarial Attacks. KDD 2019. http://pengcui.thumedialab.com/papers/RGCN.pdf

Author's Tensorflow implemention: https://github.com/thumanlab/nrlweb/tree/master/static/assets/download

class GGCL_D(*in_features*, *out_features*, *dropout*)
Graph Gaussian Convolution Layer (GGCL) when the input is distribution

class GGCL_F(*in_features*, *out_features*, *dropout=0.6*)
Graph Gaussian Convolution Layer (GGCL) when the input is feature

class GaussianConvolution(*in_features*, *out_features*)
[Deprecated] Alternative gaussion convolution layer.

class RGCN(*nnodes*, *nfeat*, *nhid*, *nclass*, *gamma=1.0*, *beta1=0.0005*, *beta2=0.0005*, *lr=0.01*, *dropout=0.6*, *device='cpu'*)
Robust Graph Convolutional Networks Against Adversarial Attacks. KDD 2019.

Parameters

- **nnodes** (*int*) – number of nodes in the input grpah
- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **gamma** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **beta1** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **beta2** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **lr** (*float*) – learning rate for GCN
- **dropout** (*float*) – dropout rate for GCN
- **device** (*str*) – 'cpu' or 'cuda'.

fit(*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *train_iters=200*, *verbose=True*, ***kwargs*)
Train RGCN.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **train_iters** (*int*) – number of training epochs
- **verbose** (*bool*) – whether to show verbose logs

Examples

We can first load dataset and then train RGCN.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import RGCN
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = RGCN(nnodes=perturbed_adj.shape[0], nfeat=features.shape[1],
```

(continues on next page)

(continued from previous page)

```
                nclass=labels.max()+1, nhid=32, device='cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val,
              train_iters=200, verbose=True)
>>> model.test(idx_test)
```

predict()

Returns output (log probabilities) of RGCN

Return type torch.FloatTensor

test(*idx_test*)
Evaluate the peformance on test set

8.6.8 Module contents

class GCN(*nfeat*, *nhid*, *nclass*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device=None*)
2 Layer Graph Convolutional Network.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – ‘cpu’ or ‘cuda’.

Examples

We can first load dataset and then train GCN.

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GCN
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> gcn = GCN(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
>>> gcn = gcn.to('cpu')
>>> gcn.fit(features, adj, labels, idx_train) # train without earlystopping
```

(continues on next page)

(continued from previous page)

```
>>> gcn.fit(features, adj, labels, idx_train, idx_val, patience=30) # train with␣
↪earlystopping
>>> gcn.test(idx_test)
```

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *train_iters=200*, *initialize=True*, *verbose=False*, *normalize=True*, *patience=500*, ***kwargs*)
Train the gcn model, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs
- **normalize** (*bool*) – whether to normalize the input adjacency matrix.
- **patience** (*int*) – patience for early stopping, only valid when *idx_val* is given

initialize()
Initialize parameters of GCN.

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCN

Return type torch.FloatTensor

test (*idx_test*)
Evaluate GCN performance on test set.

Parameters **idx_test** – node testing indices

class GCNSVD (*nfeat*, *nhid*, *nclass*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device='cpu'*)
GCNSVD is a 2 Layer Graph Convolutional Network with Truncated SVD as preprocessing. See more details in All You Need Is Low (Rank): Defending Against Adversarial Attacks on Graphs, https://dl.acm.org/doi/abs/10.1145/3336191.3371789.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units

- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GCNSVD.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import GCNSVD
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = GCNSVD(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu').to('cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val, k=20)
```

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *k=50*, *train_iters=200*, *initialize=True*, *verbose=True*, ***kwargs*)

First perform rank-k approximation of adjacency matrix via truncated SVD, and then train the gcn model on the processed graph, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **k** (*int*) – number of singular values and vectors to compute.
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCNSVD

Return type torch.FloatTensor

truncatedSVD (*data*, *k=50*)
Truncated SVD on input data.

Parameters

- **data** – input matrix to be decomposed
- **k** (*int*) – number of singular values and vectors to compute.

Returns reconstructed matrix.

Return type numpy.array

class GCNJaccard (*nfeat*, *nhid*, *nclass*, *binary_feature=True*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_relu=True*, *with_bias=True*, *device='cpu'*)
GCNJaccard first preprocesses input graph via droppining dissimilar edges and train a GCN based on the processed graph. See more details in Adversarial Examples on Graph Data: Deep Insights into Attack and Defense, https://arxiv.org/pdf/1903.01610.pdf.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_relu** (*bool*) – whether to use relu activation function. If False, GCN will be linearized.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GCNJaccard.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import GCNJaccard
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
```

(continues on next page)

(continued from previous page)

```
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = GCNJaccard(nfeat=features.shape[1],
          nhid=16,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu').to('cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val, threshold=0.03)
```

drop_dissimilar_edges (*features*, *adj*, *metric='similarity'*)
Drop dissimilar edges.(Faster version using numba)

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *threshold=0.01*, *train_iters=200*, *initialize=True*, *verbose=True*, ***kwargs*)
First drop dissimilar edges with similarity smaller than given threshold and then train the gcn model on the processed graph. When idx_val is not None, pick the best model according to the validation loss.

Parameters

- **features** – node features. The format can be numpy.array or scipy matrix
- **adj** – the adjacency matrix.
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **threshold** (*float*) – similarity threshold for dropping edges. If two connected nodes with similarity smaller than threshold, the edge between them will be removed.
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs

predict (*features=None*, *adj=None*)
By default, the inputs should be unnormalized adjacency

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCNJaccard

Return type torch.FloatTensor

class RGCN (*nnodes*, *nfeat*, *nhid*, *nclass*, *gamma=1.0*, *beta1=0.0005*, *beta2=0.0005*, *lr=0.01*, *dropout=0.6*, *device='cpu'*)
Robust Graph Convolutional Networks Against Adversarial Attacks. KDD 2019.

Parameters

- **nnodes** (*int*) – number of nodes in the input grpah

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **gamma** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **beta1** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **beta2** (*float*) – hyper-parameter for RGCN. See more details in the paper.
- **lr** (*float*) – learning rate for GCN
- **dropout** (*float*) – dropout rate for GCN
- **device** (*str*) – 'cpu' or 'cuda'.

fit (*features*, *adj*, *labels*, *idx_train*, *idx_val=None*, *train_iters=200*, *verbose=True*, ***kwargs*)
Train RGCN.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices. If not given (None), GCN training process will not adpot early stopping
- **train_iters** (*int*) – number of training epochs
- **verbose** (*bool*) – whether to show verbose logs

Examples

We can first load dataset and then train RGCN.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import RGCN
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> # train defense model
>>> model = RGCN(nnodes=perturbed_adj.shape[0], nfeat=features.shape[1],
                 nclass=labels.max()+1, nhid=32, device='cpu')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val,
              train_iters=200, verbose=True)
>>> model.test(idx_test)
```

predict()

Returns output (log probabilities) of RGCN

Return type torch.FloatTensor

test(*idx_test*)
Evaluate the peformance on test set

class ProGNN(*model*, *args*, *device*)
ProGNN (Properties Graph Neural Network). See more details in Graph Structure Learning for Robust Graph Neural Networks, KDD 2020, https://arxiv.org/abs/2005.10203.

Parameters

- **model** – model: The backbone GNN model in ProGNN
- **args** – model configs
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

See details in https://github.com/ChandlerBang/Pro-GNN.

fit(*features*, *adj*, *labels*, *idx_train*, *idx_val*, ***kwargs*)
Train Pro-GNN.

Parameters

- **features** – node features
- **adj** – the adjacency matrix. The format could be torch.tensor or scipy matrix
- **labels** – node labels
- **idx_train** – node training indices
- **idx_val** – node validation indices

test(*features*, *labels*, *idx_test*)
Evaluate the performance of ProGNN on test set

class GraphConvolution(*in_features*, *out_features*, *with_bias=True*)
Simple GCN layer, similar to https://github.com/tkipf/pygcn

forward(*input*, *adj*)
Graph Convolutional Layer forward function

class GGCL_F(*in_features*, *out_features*, *dropout=0.6*)
Graph Gaussian Convolution Layer (GGCL) when the input is feature

class GGCL_D(*in_features*, *out_features*, *dropout*)
Graph Gaussian Convolution Layer (GGCL) when the input is distribution

class GAT(*nfeat*, *nhid*, *nclass*, *heads=8*, *output_heads=1*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_bias=True*, *device=None*)
2 Layer Graph Attention Network based on pytorch geometric.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **heads** (*int*) – number of attention heads
- **output_heads** (*int*) – number of attention output heads
- **dropout** (*float*) – dropout rate for GAT

- **lr** (*float*) – learning rate for GAT
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_bias** (*bool*) – whether to include bias term in GAT weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train GAT.

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import GAT
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> gat = GAT(nfeat=features.shape[1],
          nhid=8, heads=8,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
>>> gat = gat.to('cpu')
>>> pyg_data = Dpr2Pyg(data) # convert deeprobust dataset to pyg dataset
>>> gat.fit(pyg_data, patience=100, verbose=True) # train with earlystopping
```

fit(*pyg_data*, *train_iters=1000*, *initialize=True*, *verbose=False*, *patience=100*, ***kwargs*)
Train the GAT model, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **pyg_data** – pytorch geometric dataset object
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs
- **patience** (*int*) – patience for early stopping, only valid when *idx_val* is given

initialize()
Initialize parameters of GAT.

predict()

Returns output (log probabilities) of GAT

Return type torch.FloatTensor

test()
Evaluate GAT performance on test set.

Parameters **idx_test** – node testing indices

train_with_early_stopping(*train_iters*, *patience*, *verbose*)
early stopping based on the validation loss

class ChebNet(*nfeat*, *nhid*, *nclass*, *num_hops=3*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *with_bias=True*, *device=None*)
2 Layer ChebNet based on pytorch geometric.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **num_hops** (*int*) – number of hops in ChebConv
- **dropout** (*float*) – dropout rate for ChebNet
- **lr** (*float*) – learning rate for ChebNet
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_bias** (*bool*) – whether to include bias term in ChebNet weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train ChebNet.

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import ChebNet
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> cheby = ChebNet(nfeat=features.shape[1],
          nhid=16, num_hops=3,
          nclass=labels.max().item() + 1,
          dropout=0.5, device='cpu')
>>> cheby = cheby.to('cpu')
>>> pyg_data = Dpr2Pyg(data) # convert deeprobust dataset to pyg dataset
>>> cheby.fit(pyg_data, patience=10, verbose=True) # train with earlystopping
```

fit(*pyg_data*, *train_iters=200*, *initialize=True*, *verbose=False*, *patience=500*, ***kwargs*)
Train the ChebNet model, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **pyg_data** – pytorch geometric dataset object
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs
- **patience** (*int*) – patience for early stopping, only valid when *idx_val* is given

initialize()
Initialize parameters of ChebNet.

predict()

Returns output (log probabilities) of ChebNet

Return type torch.FloatTensor

test()
Evaluate ChebNet performance on test set.

Parameters **idx_test** – node testing indices

train_with_early_stopping(*train_iters*, *patience*, *verbose*)
early stopping based on the validation loss

class SGC(*nfeat*, *nclass*, *K=3*, *cached=True*, *lr=0.01*, *weight_decay=0.0005*, *with_bias=True*, *device=None*)
SGC based on pytorch geometric. Simplifying Graph Convolutional Networks.

Parameters

- **nfeat** (*int*) – size of input feature dimension
- **nclass** (*int*) – size of output dimension
- **K** (*int*) – number of propagation in SGC
- **cached** (*bool*) – whether to set the cache flag in SGConv
- **lr** (*float*) – learning rate for SGC
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_bias** (*bool*) – whether to include bias term in SGC weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train SGC.

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.defense import SGC
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> sgc = SGC(nfeat=features.shape[1], K=3, lr=0.1,
          nclass=labels.max().item() + 1, device='cuda')
>>> sgc = sgc.to('cuda')
>>> pyg_data = Dpr2Pyg(data) # convert deeprobust dataset to pyg dataset
>>> sgc.fit(pyg_data, train_iters=200, patience=200, verbose=True) # train with␣
↪earlystopping
```

fit(*pyg_data*, *train_iters=200*, *initialize=True*, *verbose=False*, *patience=500*, ***kwargs*)
Train the SGC model, when idx_val is not None, pick the best model according to the validation loss.

Parameters

- **pyg_data** – pytorch geometric dataset object
- **train_iters** (*int*) – number of training epochs
- **initialize** (*bool*) – whether to initialize parameters before training
- **verbose** (*bool*) – whether to show verbose logs
- **patience** (*int*) – patience for early stopping, only valid when *idx_val* is given

initialize()
Initialize parameters of SGC.

predict()

Returns output (log probabilities) of SGC

Return type torch.FloatTensor

test()
Evaluate SGC performance on test set.

Parameters **idx_test** – node testing indices

train_with_early_stopping(*train_iters*, *patience*, *verbose*)
early stopping based on the validation loss

class SimPGCN(*nnodes*, *nfeat*, *nhid*, *nclass*, *dropout=0.5*, *lr=0.01*, *weight_decay=0.0005*, *lambda_=5*, *gamma=0.1*, *bias_init=0*, *with_bias=True*, *device=None*)

SimP-GCN: Node similarity preserving graph convolutional networks. https://arxiv.org/abs/2011.09643

Parameters

- **nnodes** (*int*) – number of nodes in the input grpah
- **nfeat** (*int*) – size of input feature dimension
- **nhid** (*int*) – number of hidden units
- **nclass** (*int*) – size of output dimension
- **lambda** (*float*) – coefficients for SSL loss in SimP-GCN
- **gamma** (*float*) – coefficients for adaptive learnable self-loops
- **bias_init** (*float*) – bias init for the score
- **dropout** (*float*) – dropout rate for GCN
- **lr** (*float*) – learning rate for GCN
- **weight_decay** (*float*) – weight decay coefficient (l2 normalization) for GCN. When *with_relu* is True, *weight_decay* will be set to 0.
- **with_bias** (*bool*) – whether to include bias term in GCN weights.
- **device** (*str*) – 'cpu' or 'cuda'.

Examples

We can first load dataset and then train SimPGCN.

See the detailed hyper-parameter setting in https://github.com/ChandlerBang/SimP-GCN.

```
>>> from deeprobust.graph.data import PrePtbDataset, Dataset
>>> from deeprobust.graph.defense import SimPGCN
>>> # load clean graph data
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # load perturbed graph data
>>> perturbed_data = PrePtbDataset(root='/tmp/', name='cora')
>>> perturbed_adj = perturbed_data.adj
>>> model = SimPGCN(nnodes=features.shape[0], nfeat=features.shape[1],
    nhid=16, nclass=labels.max()+1, device='cuda')
>>> model = model.to('cuda')
>>> model.fit(features, perturbed_adj, labels, idx_train, idx_val, train_
↪iters=200, verbose=True)
>>> model.test(idx_test)
```

initialize()
Initialize parameters of SimPGCN.

myforward(*fea*, *adj*)
output embedding and log_softmax

predict(*features=None*, *adj=None*)
By default, the inputs should be unnormalized data

Parameters

- **features** – node features. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.
- **adj** – adjcency matrix. If *features* and *adj* are not given, this function will use previous stored *features* and *adj* from training to make predictions.

Returns output (log probabilities) of GCN

Return type torch.FloatTensor

test(*idx_test*)
Evaluate GCN performance on test set.

Parameters **idx_test** – node testing indices

class Node2Vec
node2vec: Scalable Feature Learning for Networks. KDD'15. To use this model, you need to "pip install node2vec" first.

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import NodeEmbeddingAttack
>>> from deeprobust.graph.defense import Node2Vec
>>> data = Dataset(root='/tmp/', name='cora_ml', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # set up attack model
>>> attacker = NodeEmbeddingAttack()
>>> attacker.attack(adj, attack_type="remove", n_perturbations=1000)
>>> modified_adj = attacker.modified_adj
>>> print("Test Node2vec on clean graph")
>>> model = Node2Vec()
>>> model.fit(adj)
>>> model.evaluate_node_classification(labels, idx_train, idx_test)
>>> print("Test Node2vec on attacked graph")
>>> model = Node2Vec()
>>> model.fit(modified_adj)
>>> model.evaluate_node_classification(labels, idx_train, idx_test)
```

node2vec(*adj*, *embedding_dim=64*, *walk_length=30*, *walks_per_node=10*, *workers=8*, *window_size=10*, *num_neg_samples=1*, *p=4*, *q=1*)
Compute Node2Vec embeddings for the given graph.

Parameters

- **adj** (*sp.csr_matrix, shape [n_nodes, n_nodes]*) – Adjacency matrix of the graph
- **embedding_dim** (*int, optional*) – Dimension of the embedding

- **walks_per_node** (*int, optional*) – Number of walks sampled from each node
- **walk_length** (*int, optional*) – Length of each random walk
- **workers** (*int, optional*) – Number of threads (see gensim.models.Word2Vec process)
- **window_size** (*int, optional*) – Window size (see gensim.models.Word2Vec)
- **num_neg_samples** (*int, optional*) – Number of negative samples (see gensim.models.Word2Vec)
- **p** (*float*) – The hyperparameter p in node2vec
- **q** (*float*) – The hyperparameter q in node2vec

class DeepWalk(*type='skipgram'*)

DeepWalk: Online Learning of Social Representations. KDD'14. The implementation is modified from https://github.com/abojchevski/node_embedding_attack

Examples

```
>>> from deeprobust.graph.data import Dataset
>>> from deeprobust.graph.global_attack import NodeEmbeddingAttack
>>> from deeprobust.graph.defense import DeepWalk
>>> data = Dataset(root='/tmp/', name='cora_ml', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
>>> # set up attack model
>>> attacker = NodeEmbeddingAttack()
>>> attacker.attack(adj, attack_type="remove", n_perturbations=1000)
>>> modified_adj = attacker.modified_adj
>>> print("Test DeepWalk on clean graph")
>>> model = DeepWalk()
>>> model.fit(adj)
>>> model.evaluate_node_classification(labels, idx_train, idx_test)
>>> print("Test DeepWalk on attacked graph")
>>> model.fit(modified_adj)
>>> model.evaluate_node_classification(labels, idx_train, idx_test)
>>> print("Test DeepWalk SVD")
>>> model = DeepWalk(type="svd")
>>> model.fit(modified_adj)
>>> model.evaluate_node_classification(labels, idx_train, idx_test)
```

deepwalk_skipgram(*adj*, *embedding_dim=64*, *walk_length=80*, *walks_per_node=10*, *workers=8*, *window_size=10*, *num_neg_samples=1*)

Compute DeepWalk embeddings for the given graph using the skip-gram formulation.

Parameters

- **adj** (*sp.csr_matrix, shape [n_nodes, n_nodes]*) – Adjacency matrix of the graph
- **embedding_dim** (*int, optional*) – Dimension of the embedding
- **walks_per_node** (*int, optional*) – Number of walks sampled from each node
- **walk_length** (*int, optional*) – Length of each random walk
- **workers** (*int, optional*) – Number of threads (see gensim.models.Word2Vec process)

- **window_size** (*int, optional*) – Window size (see gensim.models.Word2Vec)
- **num_neg_samples** (*int, optional*) – Number of negative samples (see gensim.models.Word2Vec)

deepwalk_svd(*adj*, *window_size=10*, *embedding_dim=64*, *num_neg_samples=1*, *sparse=True*)

Compute DeepWalk embeddings for the given graph using the matrix factorization formulation. adj: sp.csr_matrix, shape [n_nodes, n_nodes]

Adjacency matrix of the graph

window_size: int Size of the window

embedding_dim: int Size of the embedding

num_neg_samples: int Number of negative samples

sparse: bool Whether to perform sparse operations

Returns Embedding matrix.

Return type np.ndarray, shape [num_nodes, embedding_dim]

svd_embedding(*x*, *embedding_dim*, *sparse=False*)

Computes an embedding by selection the top (embedding_dim) largest singular-values/vectors. :param x: sp.csr_matrix or np.ndarray

The matrix that we want to embed

Parameters

- **embedding_dim** – int Dimension of the embedding
- **sparse** – bool Whether to perform sparse operations

Returns np.ndarray, shape [?, embedding_dim], np.ndarray, shape [?, embedding_dim] Embedding matrices.

8.7 deeprobust.graph.data package

8.7.1 Submodules

8.7.2 deeprobust.graph.data.attacked_data module

class PrePtbDataset(*root*, *name*, *attack_method='meta'*, *ptb_rate=0.05*)

Dataset class manages pre-attacked adjacency matrix on different datasets. Note metattack is generated by deeprobust/graph/global_attack/metattack.py. While PrePtbDataset provides pre-attacked graph generate by Zugner, https://github.com/danielzuegner/gnn-meta-attack. The attacked graphs are downloaded from https://github.com/ChandlerBang/Pro-GNN/tree/master/meta.

Parameters

- **root** – root directory where the dataset should be saved.
- **name** – dataset name. It can be choosen from ['cora', 'citeseer', 'polblogs', 'pubmed']
- **attack_method** – currently this class only support metattack and nettack. Note 'meta', 'metattack' or 'mettack' will be interpreted as the same attack.
- **seed** – random seed for splitting training/validation/test.

Examples

```
>>> from deeprobust.graph.data import Dataset, PrePtbDataset
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> # Load meta attacked data
>>> perturbed_data = PrePtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='meta',
                        ptb_rate=0.05)
>>> perturbed_adj = perturbed_data.adj
>>> # Load nettacked data
>>> perturbed_data = PrePtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='nettack',
                        ptb_rate=1.0)
>>> perturbed_adj = perturbed_data.adj
>>> target_nodes = perturbed_data.target_nodes
```

get_target_nodes()
Get target nodes incides, which is the nodes with degree > 10 in the test set.

class PtbDataset(*root*, *name*, *attack_method='mettack'*)
Dataset class manages pre-attacked adjacency matrix on different datasets. Currently only support metattack under 5% perturbation. Note metattack is generated by deeprobust/graph/global_attack/metattack.py. While PrePtbDataset provides pre-attacked graph generate by Zugner, https://github.com/danielzuegner/gnn-meta-attack. The attacked graphs are downloaded from https://github.com/ChandlerBang/pytorch-gnn-meta-attack/tree/master/pre-attacked.

Parameters

- **root** – root directory where the dataset should be saved.
- **name** – dataset name. It can be choosen from ['cora', 'citeseer', 'cora_ml', 'polblogs', 'pubmed']
- **attack_method** – currently this class only support metattack. User can pass 'meta', 'metattack' or 'mettack' since all of them will be interpreted as the same attack.
- **seed** – random seed for splitting training/validation/test.

Examples

```
>>> from deeprobust.graph.data import Dataset, PtbDataset
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> perturbed_data = PtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='meta')
>>> perturbed_adj = perturbed_data.adj
```

8.7.3 deeprobust.graph.data.dataset module

class Dataset(*root*, *name*, *setting='nettack'*, *seed=None*, *require_mask=False*)
Dataset class contains four citation network datasets "cora", "cora-ml", "citeseer" and "pubmed", and one blog dataset "Polblogs". Datasets "ACM", "BlogCatalog", "Flickr", "UAI", "Flickr" are also available. See

more details in https://github.com/DSE-MSU/DeepRobust/tree/master/deeprobust/graph#supported-datasets. The 'cora', 'cora-ml', 'polblogs' and 'citeseer' are downloaded from https://github.com/danielzuegner/gnn-meta-attack/tree/master/data, and 'pubmed' is from https://github.com/tkipf/gcn/tree/master/gcn/data.

Parameters

- **root** (*string*) – root directory where the dataset should be saved.
- **name** (*string*) – dataset name, it can be chosen from ['cora', 'citeseer', 'cora_ml', 'polblogs', 'pubmed', 'acm', 'blogcatalog', 'uai', 'flickr']
- **setting** (*string*) – there are two data splits settings. It can be chosen from ['nettack', 'gcn', 'prognn'] The 'nettack' setting follows nettack paper where they select the largest connected components of the graph and use 10%/10%/80% nodes for training/validation/test . The 'gcn' setting follows gcn paper where they use the full graph and 20 samples in each class for traing, 500 nodes for validation, and 1000 nodes for test. (Note here 'netack' and 'gcn' setting do not provide fixed split, i.e., different random seed would return different data splits)
- **seed** (*int*) – random seed for splitting training/validation/test.
- **require_mask** (*bool*) – setting require_mask True to get training, validation and test mask (self.train_mask, self.val_mask, self.test_mask)

Examples

We can first create an instance of the Dataset class and then take out its attributes.

```
>>> from deeprobust.graph.data import Dataset
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
```

download_npz()
Download adjacen matrix npz file from self.url.

get_prognn_splits()
Get target nodes incides, which is the nodes with degree > 10 in the test set.

get_train_val_test()
Get training, validation, test splits according to self.setting (either 'nettack' or 'gcn').

largest_connected_components(*adj*, *n_components=1*)
Select k largest connected components.

Parameters

- **adj** (*scipy.sparse.csr_matrix*) – input adjacency matrix
- **n_components** (*int*) – n largest connected components we want to select

8.7.4 Module contents

class Dataset(*root*, *name*, *setting='nettack'*, *seed=None*, *require_mask=False*)
Dataset class contains four citation network datasets "cora", "cora-ml", "citeseer" and "pubmed", and one blog dataset "Polblogs". Datasets "ACM", "BlogCatalog", "Flickr", "UAI", "Flickr" are also available. See more details in https://github.com/DSE-MSU/DeepRobust/tree/master/deeprobust/graph#supported-datasets. The 'cora', 'cora-ml', 'polblogs' and 'citeseer' are downloaded from https://github.com/danielzuegner/gnn-meta-attack/tree/master/data, and 'pubmed' is from https://github.com/tkipf/gcn/tree/master/gcn/data.

Parameters

- **root** (*string*) – root directory where the dataset should be saved.
- **name** (*string*) – dataset name, it can be chosen from ['cora', 'citeseer', 'cora_ml', 'polblogs', 'pubmed', 'acm', 'blogcatalog', 'uai', 'flickr']
- **setting** (*string*) – there are two data splits settings. It can be chosen from ['nettack', 'gcn', 'prognn'] The 'nettack' setting follows nettack paper where they select the largest connected components of the graph and use 10%/10%/80% nodes for training/validation/test . The 'gcn' setting follows gcn paper where they use the full graph and 20 samples in each class for traing, 500 nodes for validation, and 1000 nodes for test. (Note here 'netack' and 'gcn' setting do not provide fixed split, i.e., different random seed would return different data splits)
- **seed** (*int*) – random seed for splitting training/validation/test.
- **require_mask** (*bool*) – setting require_mask True to get training, validation and test mask (self.train_mask, self.val_mask, self.test_mask)

Examples

We can first create an instance of the Dataset class and then take out its attributes.

```
>>> from deeprobust.graph.data import Dataset
>>> data = Dataset(root='/tmp/', name='cora', seed=15)
>>> adj, features, labels = data.adj, data.features, data.labels
>>> idx_train, idx_val, idx_test = data.idx_train, data.idx_val, data.idx_test
```

download_npz()
Download adjacen matrix npz file from self.url.

get_prognn_splits()
Get target nodes incides, which is the nodes with degree > 10 in the test set.

get_train_val_test()
Get training, validation, test splits according to self.setting (either 'nettack' or 'gcn').

largest_connected_components(*adj*, *n_components=1*)
Select k largest connected components.

Parameters

- **adj** (*scipy.sparse.csr_matrix*) – input adjacency matrix
- **n_components** (*int*) – n largest connected components we want to select

class PtbDataset(*root*, *name*, *attack_method='mettack'*)
Dataset class manages pre-attacked adjacency matrix on different datasets. Currently only support metattack under 5% perturbation. Note metattack is generated by deeprobust/graph/global_attack/metattack.py. While PrePtbDataset provides pre-attacked graph generate by Zugner, https://github.com/danielzuegner/gnn-meta-attack. The attacked graphs are downloaded from https://github.com/ChandlerBang/pytorch-gnn-meta-attack/tree/master/pre-attacked.

Parameters

- **root** – root directory where the dataset should be saved.
- **name** – dataset name. It can be choosen from ['cora', 'citeseer', 'cora_ml', 'polblogs', 'pubmed']

- **attack_method** – currently this class only support metattack. User can pass 'meta', 'metattack' or 'mettack' since all of them will be interpreted as the same attack.
- **seed** – random seed for splitting training/validation/test.

Examples

```
>>> from deeprobust.graph.data import Dataset, PtbDataset
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> perturbed_data = PtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='meta')
>>> perturbed_adj = perturbed_data.adj
```

class PrePtbDataset(*root*, *name*, *attack_method='meta'*, *ptb_rate=0.05*)

Dataset class manages pre-attacked adjacency matrix on different datasets. Note metattack is generated by deeprobust/graph/global_attack/metattack.py. While PrePtbDataset provides pre-attacked graph generate by Zugner, https://github.com/danielzuegner/gnn-meta-attack. The attacked graphs are downloaded from https://github.com/ChandlerBang/Pro-GNN/tree/master/meta.

Parameters

- **root** – root directory where the dataset should be saved.
- **name** – dataset name. It can be choosen from ['cora', 'citeseer', 'polblogs', 'pubmed']
- **attack_method** – currently this class only support metattack and nettack. Note 'meta', 'metattack' or 'mettack' will be interpreted as the same attack.
- **seed** – random seed for splitting training/validation/test.

Examples

```
>>> from deeprobust.graph.data import Dataset, PrePtbDataset
>>> data = Dataset(root='/tmp/', name='cora')
>>> adj, features, labels = data.adj, data.features, data.labels
>>> # Load meta attacked data
>>> perturbed_data = PrePtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='meta',
                        ptb_rate=0.05)
>>> perturbed_adj = perturbed_data.adj
>>> # Load nettacked data
>>> perturbed_data = PrePtbDataset(root='/tmp/',
                        name='cora',
                        attack_method='nettack',
                        ptb_rate=1.0)
>>> perturbed_adj = perturbed_data.adj
>>> target_nodes = perturbed_data.target_nodes
```

get_target_nodes()

Get target nodes incides, which is the nodes with degree > 10 in the test set.

class Pyg2Dpr(*pyg_data*, ***kwargs*)

Convert pytorch geometric data (tensor, edge_index) to deeprobust data (sparse matrix)

Parameters **pyg_data** – data instance of class from pytorch geometric dataset

Examples

We can first create an instance of the Dataset class and convert it to pytorch geometric data format and then convert it back to Dataset class.

```
>>> from deeprobust.graph.data import Dataset, Dpr2Pyg, Pyg2Dpr
>>> data = Dataset(root='/tmp/', name='cora')
>>> pyg_data = Dpr2Pyg(data)
>>> print(pyg_data)
>>> print(pyg_data[0])
>>> dpr_data = Pyg2Dpr(pyg_data)
>>> print(dpr_data.adj)
```

class Dpr2Pyg(*dpr_data*, *transform=None*, ***kwargs*)

Convert deeprobust data (sparse matrix) to pytorch geometric data (tensor, edge_index)

Parameters

- **dpr_data** – data instance of class from deeprobust.graph.data, e.g., deeprobust.graph.data.Dataset, deeprobust.graph.data.PtbDataset, deeprobust.graph.data.PrePtbDataset
- **transform** – A function/transform that takes in an object and returns a transformed version. The data object will be transformed before every access. For example, you can use torch_geometric.transforms.NormalizeFeatures()

Examples

We can first create an instance of the Dataset class and convert it to pytorch geometric data format.

```
>>> from deeprobust.graph.data import Dataset, Dpr2Pyg
>>> data = Dataset(root='/tmp/', name='cora')
>>> pyg_data = Dpr2Pyg(data)
>>> print(pyg_data)
>>> print(pyg_data[0])
```

update_edge_index(*adj*)

This is an inplace operation to substitute the original edge_index with adj.nonzero()

Parameters **adj** (*sp.csr_matrix*) – update the original adjacency into adj (by change edge_index)

class AmazonPyg(*root*, *name*, *transform=None*, *pre_transform=None*, ***kwargs*)

Amazon-Computers and Amazon-Photo datasets loaded from pytorch geomtric; the way we split the dataset follows Towards Deeper Graph Neural Networks (https://github.com/mengliu1998/DeeperGNN/blob/master/DeeperGNN/train_eval.py). Specifically, 20 * num_classes labels for training, 30 * num_classes labels for validation, rest labels for testing.

Parameters

- **root** (*string*) – root directory where the dataset should be saved.
- **name** (*string*) – dataset name, it can be choosen from ['computers', 'photo']
- **transform** – A function/transform that takes in an torch_geometric.data.Data object and returns a transformed version. The data object will be transformed before every access. (default: None)

- **pre_transform** – A function/transform that takes in an torch_geometric.data.Data object and returns a transformed version. The data object will be transformed before being saved to disk.

Examples

We can directly load Amazon dataset from deeprobust in the format of pyg.

```
>>> from deeprobust.graph.data import AmazonPyg
>>> computers = AmazonPyg(root='/tmp', name='computers')
>>> print(computers)
>>> print(computers[0])
>>> photo = AmazonPyg(root='/tmp', name='photo')
>>> print(photo)
>>> print(photo[0])
```

class CoauthorPyg(*root*, *name*, *transform=None*, *pre_transform=None*, ***kwargs*)

Coauthor-CS and Coauthor-Physics datasets loaded from pytorch geomtric; the way we split the dataset follows Towards Deeper Graph Neural Networks (https://github.com/mengliu1998/DeeperGNN/blob/master/DeeperGNN/train_eval.py). Specifically, 20 * num_classes labels for training, 30 * num_classes labels for validation, rest labels for testing.

Parameters

- **root** (*string*) – root directory where the dataset should be saved.
- **name** (*string*) – dataset name, it can be choosen from ['cs', 'physics']
- **transform** – A function/transform that takes in an torch_geometric.data.Data object and returns a transformed version. The data object will be transformed before every access. (default: None)
- **pre_transform** – A function/transform that takes in an torch_geometric.data.Data object and returns a transformed version. The data object will be transformed before being saved to disk.

Examples

We can directly load Coauthor dataset from deeprobust in the format of pyg.

```
>>> from deeprobust.graph.data import CoauthorPyg
>>> cs = CoauthorPyg(root='/tmp', name='cs')
>>> print(cs)
>>> print(cs[0])
>>> physics = CoauthorPyg(root='/tmp', name='physics')
>>> print(physics)
>>> print(physics[0])
```

CHAPTER 9

Indices and tables

- modindex
- search

Python Module Index

d

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V