Project Report: Semi-supervised Community Detection with Graph Convolutional Networks

Kaikai Zhao

April 29, 2019

1 Introduction

There are lots of large networks in our real-world systems, like social media networks, citation networks, and protein interaction networks, etc. In general, many of them consist of community structures. Hence, detecting the module structures in these networks is meaningful for characterizing the organizational structures and understanding complex systems. In fact, labeling the community structures for large networks is quite time-consuming and laborious. However, labeling a small portion of large networks is possible. Actually, sometimes labels of a few samples are available. It will be of significance to make use of the samples with available labels to learn the data distribution and detect the community structure of large networks.

In recent years, a number of approaches have been developed to address this issue. Yang et al. [2016a] attempts to reconstruct a modularity matrix on network substructures via clustering latent representations learned by deep nonlinear reconstruction model. Some researchers use graph convolutional networks to learn the characteristics of large graphs and achieve great performance (Defferrard et al. [2016];Yang et al. [2016b];Kipf and Welling [2016]). Kipf and Welling [2016] presents a scalable approach for semi-supervised learning on graph-sructured data which is based on an efficient variant of convolutional nerual networks.

2 Semi-supervised community detection based on reconstructing modularity matrix via Auto-Encoder

The first goal of this project is to reproduce the model in Yang et al. [2016a]. It proposes a modularity based community detection approach. Specifically, it attempts to reconstruct the modularity matrix via a stacked Auto-Encoder. The modularity maximization model was introduced by Newman [2004]. By defining modularity matrix $\mathbf{B} = [b_{ij}] \in \mathbb{R}^{N\times N}$ whose elements are $b_{ij} = a_{ij} - \frac{k_i k_j}{2m}$. $\frac{k_i k_j}{2m}$ is the expected number of edges between vertices i and j if edges are placed randomly, k_i is the degree of vertex i and $m = \frac{1}{2}\sum_i k_i$ is the total number of edges in the network. The i^{th} column b_i of $\mathbf{B}$ represents vertex i. The encoder maps the original data $\mathbf{B}$ to a low-dimensional embedding $\mathbf{H}$ and each column of $\mathbf{H}$, i.e. h_i represents vertex i in the latent space

$$\mathbf{h_i} = f(\mathbf{b_i}) = s(\mathbf{W}_H\mathbf{b_i} + \mathbf{d}_H) \tag{1}$$

where W_H, d_H are the parameters to be learned in the encoder and $s(\cdot)$ is an element-wise nonlinear mapping. The decoder maps the latent representation $\mathbf{H}$ back into the original data space, i.e., reconstructs the original data from the latent representation:

$$\mathbf{m_i} = g(\mathbf{h_i}) = s(\mathbf{W}_M\mathbf{h_i} + \mathbf{d}_M) \tag{2}$$

where W_M and d_M are the parameters to be learned in the decoder. Auto-Encoder aims at learning a low-dimensional nonlinear representation $\mathbf{H}$ that can best reconstruct the original data $\mathbf{B}$, i.e. minimize the difference between the original data $\mathbf{B}$ and reconstruction data $\mathbf{M}$ under parameters $\theta = \{\mathbf{W}_H, \mathbf{d}_H, \mathbf{W}_M, \mathbf{d}_M\}$

$$\begin{aligned}\hat{\theta} &= \underset{\theta}{\operatorname{argmin}} L_\theta(\mathbf{B}, \mathbf{M}) = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} L_\theta\left(\mathbf{b_i}, \mathbf{m_i}\right) \\ &= \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} L_\theta\left(\mathbf{b_i}, g\left(f\left(\mathbf{b_i}\right)\right)\right)\end{aligned} \tag{3}$$

where $L_\theta\left(\mathbf{b_i}, \mathbf{m_i}\right)$ is a distance function that measures the reconstruction error.

We define a pairwise constraint matrix $\mathbf{O} = [o_{ij}] \in \mathbb{R}_+^{N \times N}$, where $o_{ij} = 1$ if nodes i and j are known to be in the same community, or $o_{ij} = 0$ otherwise. Thus, we can write the pairwise constraints as

$$\begin{aligned}\mathcal{R}_{LSE}(\mathbf{O}, \mathbf{H}) &= \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} o_{ij} \left\|\mathbf{h}_i - \mathbf{h}_j\right\|_2^2 \\ &= \operatorname{Tr}\left(\mathbf{H}^T \mathbf{D} \mathbf{H}\right) - \operatorname{Tr}\left(\mathbf{H}^T \mathbf{O} \mathbf{H}\right) \\ &= \operatorname{Tr}\left(\mathbf{H}^T \mathbf{L} \mathbf{H}\right)\end{aligned} \tag{4}$$

where $\mathbf{D}$ is a diagonal matrix whose entries are row summation of $\mathbf{O}$, and $\mathbf{L} = \mathbf{D} - \mathbf{O}$ the graph regularization matrix (Laplacian matrix) of a priori information $\mathbf{O}$.

By incorporating the pairwise loss with the reconstruction error, we obtain the overall loss function for semi-supervised community detection,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} L_\theta(\mathbf{B}, \mathbf{M}) + \lambda \operatorname{Tr}\left(\mathbf{H}^T \mathbf{L} \mathbf{H}\right) \tag{5}$$

where λ is a tradeoff parameter between the reconstruction error and the pairwise error.

3 Semi-supervised community detection with graph convolutional networks

Since the modularity based method does not make use of the features of nodes, we try to replace the dense layer of Auto-Encoder with the graph convolutional networks(GCN) which has the identical structure to the GCN in Kipf and Welling [2016]. Therefore, our foward model takes the simple form:

$$H_1 = f(X, A) = \operatorname{Relu}(\hat{A} X W^0) \tag{6}$$

$$H_2 = f(H_1, A) = \operatorname{Relu}(\hat{A} H_1 W^1) \tag{7}$$

where $\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. A is the adjacency matrix and X is the feature matrix. W^0 and W^1 are the weight matrices.

For the semi-supervised loss, we use two types of loss functions. The first one is incorporating the pairwise constraint, i.e. in the form of Eq. (4). The second one is applying softmax activation function row-wise on the latent representations H_2, i.e. $\operatorname{softmax}(h_2(i)) = \frac{1}{Z} \exp(x(i))$ with $Z = \sum_i \exp\left(x(i)\right)$. Then we evaluate the cross-entropy error over all labeled examples as Eq. (8).

$$\mathcal{L} = -\sum_{l \in \mathcal{Y}_L} Y_l \ln Z_l \tag{8}$$

where $\mathcal{Y}_L$ is the set of the node indices that have labels.

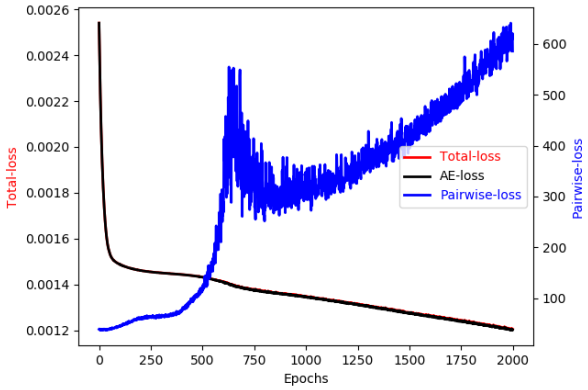

Figure 1: Loss

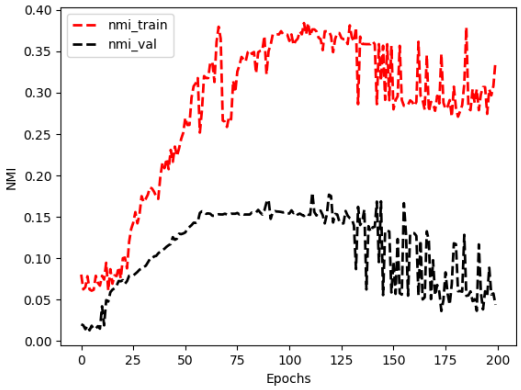

Figure 2: NMI

4 Implementations and experiments

In this section, we conducted four implementations and the corresponding experiments. Specifically, two implementations are using dense layers for the Auto-Encoder structure and two of them are using GCN. For all the experiments, we use one citation data set, i.e. Cora, to test the performance of models and 10% of labels are employed for the supervised loss. We train on the whole data set and use the remaining 90% samples to test whether the models have learned some labeling information from the samples that have labels. We use k-means to cluster the learned latent representations. Additionally, normalized mutual information is employed as the evaluation metric. The statistics of the Cora data set is in Table 2. In our report we use dataset splits provided by https://github.com/kimiyoung/planetoid. Code to reproduce our experiments is also available at https://github.com/KaikaiZhao/Graph-Analytics-Project.

Table 1: Dataset statistics

Dataset	Type	Nodes	Edges	Classes	Features
Cora	Citation network	2,708	5,429	7	1,433

4.1 Auto-Encoder + Pairwise loss

For the first implementation, the structure of Auto-Encoder is 2708-512-256-128. The tradeoff parameter λ is 5×10^{-9} and the dropout rate is 0.2. After the model is trained, we cluster the latent representations of all nodes in the validation set. We plot the total loss, reconstruction loss, and pairwise loss in Fig. 1. Also, the training NMI and validation NMI is plotted as well in Fig. 2. Finally, we report the maximum NMI on the validation set, i.e. 0.1793.

4.2 Stacked Auto-Encoder + Pairwise loss

In the second implementation, we use stacked Auto-Encoder like Yang et al. [2016a]. The difference is the way how to train the Auto-Encoder. Stacked Auto-Encoder means training the three sub-Auto-Encoder independently. In other words, after the sub-Auto-Encoder of 2708-512 is trained, we take the latent representations with the dimension of 512 as the input of the subsequent sub-Auto-Encoder. Then we train the

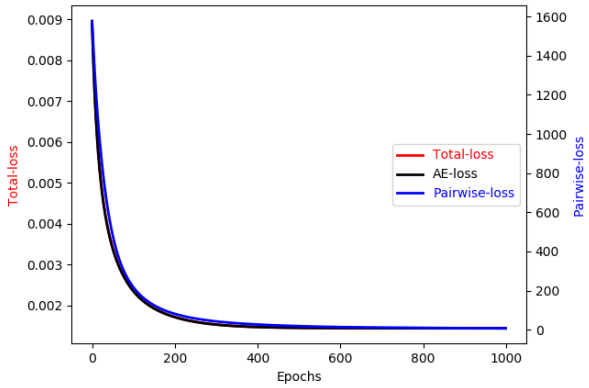

Figure 3: Loss

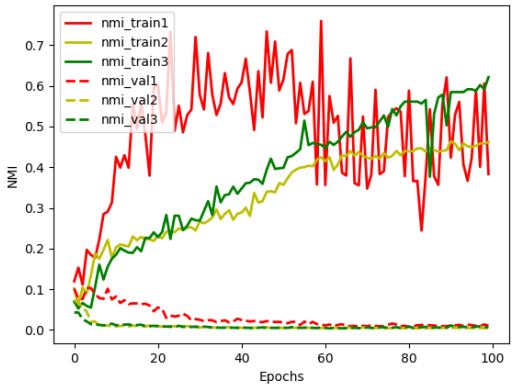

Figure 4: NMI

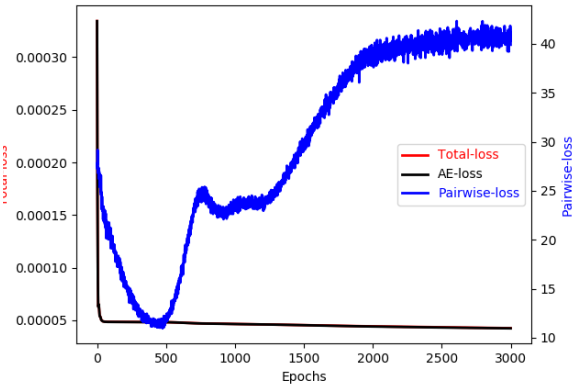

Figure 5: Loss

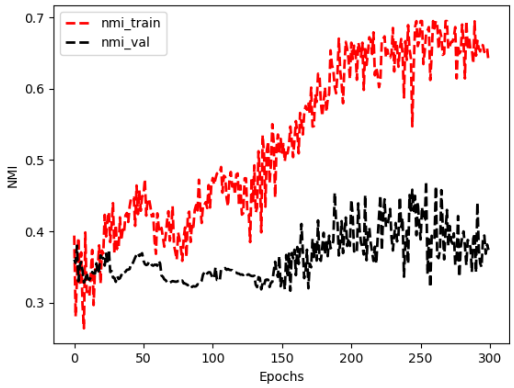

Figure 6: NMI

sub-Auto-Encoder of 512-256 and 256-128 in the same way. At last, we get a model with three sub-Auto-Encoders. We use all the latent representations of the three sub-Auto-Encoders for clustering, respectively. Also, we plot the loss and NMI during the training process in Fig. 3 and Fig. 4. However, the performance of this model is bad, i.e 0.1034, 0.0720, 0.0431, which correspond to different latent representations from the three sub-Auto-Encoders.

4.3 GCN + Pairwise loss

In this model, we replace dense layers with graph convolutional networks and the pairwise loss remains unchanged. The structure of our model is 1433-512-128. The tradeoff parameter λ is 4×10^{-9} and the dropout rate is 0.2. We present the loss and NMI during the training process in Fig. 5 and Fig. 6. We print the NMI outcome each ten epochs. The NMI on validation set is 0.4702, which is slightly better than the result in Yang et al. [2016a], i.e. 0.463.

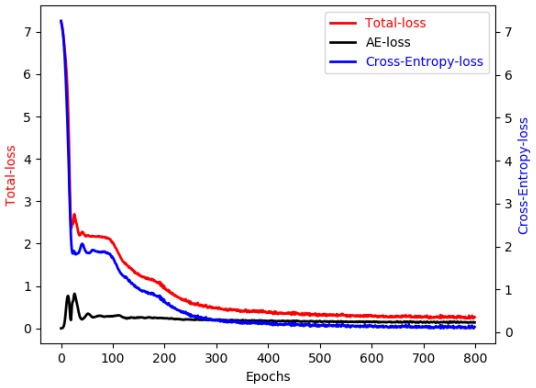

Figure 7: Loss

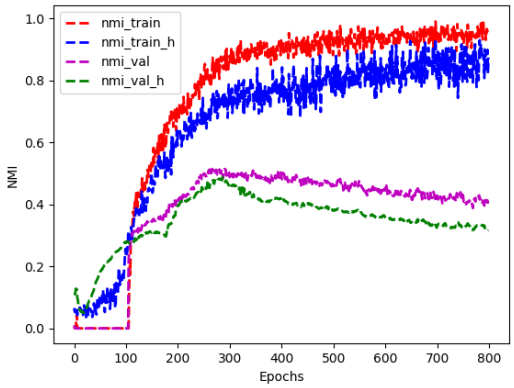

Figure 8: NMI

4.4 GCN + Cross-entropy loss

In this case, We substitute pairwise loss with the cross-entropy loss. Then the total loss is adding reconstruction loss and the cross-entropy loss. The structure of the model is 1433-32-7. We show the loss and NMI in Fig. 7 and Fig. 8. Also, we can predict the labels of all nodes directly. However, we take the learned latent vectors and the predicted labels as the input of k-means, respectively.

At last, we report the results of the four models in Table 2. For the GCN + Cross-Entropy model, we cluster the predicted labels and the learned latent representations, respectively. So we report two NMIs in the table and the number in the parenthesis denotes the result from the latent representations.

Table 2: Statistics of data sets

Models	NMI
Yang et al. [2016a]	0.463
AE+Pairwise-loss	0.1793
SAE+Pairwise-loss	0.1034
GCN+Pairwise-loss	**0.4702**
GCN+CrossEntropy-loss	**0.5140(0.4883)**

5 Conclusion and future work

As can be seen from the experimental results, the GCN models outperforms the modularity based models. That makes sense since GCN model makes use of adjacency information and features of nodes simultaneously. Both of them are powerful tools for community detection. Combining them together to analyze the characteristics of large networks is an interesting future work.

6 Acknowledgments

Thank Professor Ariful Azad for leading me into the Graph Analytics community. He is a very helpful and nice professor. When I was doing this project, I benefited a lot from the discussion with my Chinese

friends, Xifeng Guo and Zhen Cheng, who are familiar with Pytorch. Thank Xifeng and Zhen for the helpful discussion.

References

Liang Yang, Xiaochun Cao, Dongxiao He, Chuan Wang, Xiao Wang, and Weixiong Zhang. Modularity based community detection with deep learning. In *IJCAI*, volume 16, pages 2252–2258, 2016a.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. *arXiv preprint arXiv:1603.08861*, 2016b.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6): 066133, 2004.

7 Code

```
'''
When I wrote the following code, I referenced the following paper:
    1. Liang Yang, et al. Modularity based Community Detection with Deep Learning. IJCAI 2016
    2. Thomas N. Kipf, et al. Semi-supervised classification with graph convolutional networks.
        ICLR 2017

I tried to reproduce the results in [1]. Also, I referred to some online code:
    a. https://github.com/tkipf/pygcn
    b. https://github.com/ShayanPersonal/stacked-autoencoder-pytorch
    c. http://yangliang.github.io/code/DC.zip

I have given relatively detailed comments of some key lines for some potential readers.
Author: Kaikai Zhao
Email: zhaokai@iu.edu
'''

import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
import scipy.sparse as sp
from torch.autograd import Variable

def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    return labels_onehot
```

```
def
    load_data(path="C:/Users/kevin/Downloads/Coursework/GRAPH_ANALYTICS/pygcn/pygcn-master/data/cora/",
    dataset="cora"):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))

    idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                        dtype=np.dtype(str))
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    edges_ordered_flatten = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                    dtype=np.int32)
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                    dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
    adj_unnormalize = adj.copy() # copy the unnormalized adjacency matrix

    features = normalize(features)
    adj = normalize(adj + sp.eye(adj.shape[0]))

    idx_train = range(270)
    idx_val = range(270, 2707)
    idx_test = range(1777, 2707)

    features = torch.FloatTensor(np.array(features.todense()))
    labels = torch.LongTensor(np.where(labels)[1])
    adj = sparse_mx_to_torch_sparse_tensor(adj)

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)
    idx_test = torch.LongTensor(idx_test)

    return adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx

def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
```

```
    return correct / len(labels)

def sparse_mx_to_torch_sparse_tensor(sparse_mx):
    """Convert a scipy sparse matrix to a torch sparse tensor."""
    sparse_mx = sparse_mx.tocoo().astype(np.float32)
    indices = torch.from_numpy(
        np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))
    values = torch.from_numpy(sparse_mx.data)
    shape = torch.Size(sparse_mx.shape)
    return torch.sparse.FloatTensor(indices, values, shape)

# Load data
# I use Kipf's load_data() and modified it by returning an unnormalized adjacency matrix
adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize = load_data('')
adj_array = adj_unnormalize.toarray() # transform csr to dense array

# Build modularity matrix. For this part, please refer to Section 2.2 of [1]
# k_i: the degree of vertex i; total number of edges: m = 1/2 * sum_i(k_i)
# (k_i*k_j)/(2*m) describes the expected number of edges between vertices i and j if edges are
    placed randomly
# Modularity matrix b_ij = a_ij - (k_i*k_j)/(2*m)
degrees = np.array(adj_array.sum(1)) # calculate the degree of each node
degrees = degrees[:,np.newaxis] # shape: V * 1
m = 1/2 * degrees.sum() # the total number of edges in the network
B = adj_array - 1/(2*m) * degrees.dot(degrees.T) # Modularity matrix
B_tensor = torch.from_numpy(B) # convert numpy array into torch tensor

# This loss function adds the AutoEncoder loss and the pairwise loss together.
# lamd is a tradeoff hyperparameter.
class Semi_Loss(torch.nn.Module):

    def __init__(self, lamd=0.01):
        super(Semi_Loss,self).__init__()
        self.lamd = lamd

    def forward(self, AE_loss, semi_loss):
        semi_loss = AE_loss + self.lamd * semi_loss
        return semi_loss

# build Laplace matrix. For this part, please refer to Section 4 of [1]
# L = D - O
labels_train = encode_onehot(labels[idx_train].numpy())
n = idx_train.numpy().size
O = labels_train.dot(labels_train.T) - np.eye(n) # O denotes pairwise constraint matrix
L = np.diag(O.sum(1)) - O # L = D - O, D is a diagonal matrix whose entries are row summation of O
L = torch.from_numpy(L).float() # convert numpy array into torch float tensor

from sklearn.cluster import KMeans
from sklearn.metrics.cluster import normalized_mutual_info_score
# calculate normalized mutual info to measure the clustering performance
def cal_nmi(data, labels, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++').fit(data)
    pred_tr = kmeans.labels_
    nmi = normalized_mutual_info_score(labels, pred_tr, average_method='arithmetic')
    return nmi

class AutoEncoder(nn.ModuleList):
    def __init__(self):
        super(AutoEncoder, self).__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Linear(2708, 512),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(256, 128),
            nn.ReLU(),
        )
        self.decoder = nn.Sequential(
            nn.Linear(128, 256),
            nn.ReLU(), #Tanh()
            nn.Dropout(0.2),
            nn.Linear(256, 512),
            nn.ReLU(), #nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(512, 2708),
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded

B_tensor = B_tensor.cuda()
L = L.cuda()
labels = labels.cpu()

lamd = 5*10**-9
model = AutoEncoder()
model = model.cuda()
criterion = Semi_Loss(lamd)
criterion_AE = nn.MSELoss()
# criterion_AE = CrossEntropy_Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, weight_decay=0)#10**-6
epochs = 2000
totalloss_list=[];pairwise_list=[];AE_list=[];nmi_train=[];nmi_val=[]

for epoch in range(epochs):
    model.train()
#     B_tensor = Variable(B_tensor)
    H, M = model(B_tensor) # H denotes low-embedding vectors, M is the reconstruction matrix
    # For pairwise_loss, we only need to compute those which have labels.
#     Our goal is to learn the low embedding representations of all vertices via a small portion of
     vertices with labels
    pairwise = torch.mm( torch.mm(H[idx_train,:].t(), L), H[idx_train,:] ) # H'*L*H
    pairwise_loss = torch.trace(pairwise) # trace(H'*L*H)
    AE_loss = criterion_AE(M, B_tensor) # compute the reconstruction loss of AE
    total_loss = criterion(AE_loss, pairwise_loss) # total loss= recon_loss + lamd*pairwise_loss
    totalloss_list.append(total_loss.item()); pairwise_list.append(pairwise_loss.item());
         AE_list.append(AE_loss.item())
    if epoch%10 == 0:
        model.eval()
        H, M = model(B_tensor)
        H = H.data.cpu()
        nmi_train_H = cal_nmi(data=H[idx_train.numpy(),:].data.cpu().numpy(),
            labels=labels[idx_train.numpy()], n_clusters=7)
```

```
        nmi_val_H = cal_nmi(data=H[idx_val.numpy(),:].detach().numpy(),
            labels=labels[idx_val.numpy()], n_clusters=7)
        print('epoch {}: total_loss:{:.6f}|AE_loss: {:.6f}|pair_loss: {:.6f}|NMI train: {:.2f}|val:
            {:.2f}'.format
              (epoch, total_loss, AE_loss, pairwise_loss, nmi_train_H, nmi_val_H))
        model.train()
        nmi_train.append(nmi_train_H); nmi_val.append(nmi_val_H)

    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt
host = host_subplot(111)
par = host.twinx()
host.set_xlabel("Epochs")
host.set_ylabel("Total-loss")
par.set_ylabel("Pairwise-loss")
p1, = host.plot(range(epochs), totalloss_list, 'r-', lw=2, label="Total-loss")
p2, = host.plot(range(epochs), AE_list, 'k-', lw=2, label="AE-loss")
p3, = par.plot(range(epochs), np.array(pairwise_list), 'b-', lw=2, label="Pairwise-loss")
leg = plt.legend(loc='best')
host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())
par.yaxis.get_label().set_color(p3.get_color())
leg.texts[2].set_color(p3.get_color())
plt.show()

plt.plot( nmi_train, 'r-', lw=2, label='nmi_train')
plt.plot( nmi_val, 'k-', lw=2, label='nmi_val')
plt.legend(loc='upper left')
plt.show()
print(np.max(nmi_val))
```

Stacked AE + pairwise loss

```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
import scipy.sparse as sp
from torch.autograd import Variable

def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    return labels_onehot

def
     load_data(path="C:/Users/kevin/Downloads/Coursework/GRAPH_ANALYTICS/pygcn/pygcn-master/data/cora/",
     dataset="cora"):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))
```

```
    idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                        dtype=np.dtype(str))
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    edges_ordered_flatten = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32)
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
    adj_unnormalize = adj.copy() # copy the unnormalized adjacency matrix

    features = normalize(features)
    adj = normalize(adj + sp.eye(adj.shape[0]))

    idx_train = range(270)
    idx_val = range(270, 2707)
    idx_test = range(2200, 2707)

    features = torch.FloatTensor(np.array(features.todense()))
    labels = torch.LongTensor(np.where(labels)[1])
    adj = sparse_mx_to_torch_sparse_tensor(adj)

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)
    idx_test = torch.LongTensor(idx_test)

    return adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx

def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
    return correct / len(labels)

def sparse_mx_to_torch_sparse_tensor(sparse_mx):
    """Convert a scipy sparse matrix to a torch sparse tensor."""
    sparse_mx = sparse_mx.tocoo().astype(np.float32)
    indices = torch.from_numpy(
```

```
        np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))
    values = torch.from_numpy(sparse_mx.data)
    shape = torch.Size(sparse_mx.shape)
    return torch.sparse.FloatTensor(indices, values, shape)

# Load data
# I use Kipf's load_data() and modified it by returning an unnormalized adjacency matrix
adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize = load_data('')
adj_array = adj_unnormalize.toarray() # transform csr to dense array

# Build modularity matrix. For this part, please refer to Section 2.2 of [1]
# k_i: the degree of vertex i; total number of edges: m = 1/2 * sum_i(k_i)
# (k_i*k_j)/(2*m) describes the expected number of edges between vertices i and j if edges are
    placed randomly
# Modularity matrix b_ij = a_ij - (k_i*k_j)/(2*m)
degrees = np.array(adj_array.sum(1)) # calculate the degree of each node
degrees = degrees[:,np.newaxis] # shape: V * 1
m = 1/2 * degrees.sum() # the total number of edges in the network
B = adj_array - 1/(2*m) * degrees.dot(degrees.T) # Modularity matrix
B_tensor = torch.from_numpy(B) # convert numpy array into torch tensor

# This loss function adds the AutoEncoder loss and the pairwise loss together.
# lamd is a tradeoff hyperparameter.
class Semi_Loss(torch.nn.Module):

    def __init__(self, lamd=0.01):
        super(Semi_Loss,self).__init__()
        self.lamd = lamd

    def forward(self, AE_loss, semi_loss):
        semi_loss = AE_loss + self.lamd * semi_loss
        return semi_loss

# build Laplace matrix. For this part, please refer to Section 4 of [1]
# L = D - O
labels_train = encode_onehot(labels[idx_train].numpy())
n = idx_train.numpy().size
O = labels_train.dot(labels_train.T) - np.eye(n) # O denotes pairwise constraint matrix
L = np.diag(O.sum(1)) - O # L = D - O, D is a diagonal matrix whose entries are row summation of O
L = torch.from_numpy(L).float() # convert numpy array into torch float tensor

from sklearn.cluster import KMeans
from sklearn.metrics.cluster import normalized_mutual_info_score
# calculate normalized mutual info to measure the clustering performance
def cal_nmi(data, labels, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++').fit(data)
    pred_tr = kmeans.labels_
    nmi = normalized_mutual_info_score(labels, pred_tr, average_method='arithmetic')
    return nmi

class AutoEncoder(nn.Module):
    def __init__(self, input_size, output_size):
        super(AutoEncoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(input_size, output_size),
            nn.ReLU(),
        )
        self.decoder = nn.Sequential(
```

```
            nn.Linear(output_size, input_size),
        )

    def forward(self, x):
        # Train each autoencoder individually
        x = x.detach()
        y = self.encoder(x)

        return y.detach(), y

    def reconstruct(self, x):
        return self.decoder(x)

class StackedAutoEncoder(nn.Module):
    r"""
    A stacked autoencoder made from the convolutional denoising autoencoders above.
    Each autoencoder is trained independently and at the same time.
    """

    def __init__(self):
        super(StackedAutoEncoder, self).__init__()

        self.AE1 = AutoEncoder(2708, 512)
        self.AE2 = AutoEncoder(512, 256)
        self.AE3 = AutoEncoder(256, 128)

    def forward(self, x):
        a1,h1 = self.AE1(x) # a1 has been detached. h1 can be used for backpropagation
        a2,h2 = self.AE2(a1)
        a3,h3 = self.AE3(a2)

        return h1, h2, h3, self.reconstruct(a3)

    def reconstruct(self, x):
            a2_reconstruct = self.AE3.reconstruct(x)
            a1_reconstruct = self.AE2.reconstruct(a2_reconstruct)
            x_reconstruct = self.AE1.reconstruct(a1_reconstruct)
            return x_reconstruct, a1_reconstruct, a2_reconstruct

B_tensor = B_tensor.cuda()
L = L.cuda()
labels = labels.cpu()

lamd = 5*10**-8
model = StackedAutoEncoder()
model = model.cuda()
labels = labels.cpu()
criterion = Semi_Loss(lamd)
criterion_AE = nn.MSELoss()
# criterion_AE = CrossEntropy_Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, weight_decay=10**-9)#10**-6
epochs = 1000
totalloss_list=[];pairwise_list=[];AE_list=[];
nmi_train1=[]; nmi_train2=[]; nmi_train3=[];
nmi_val1=[]; nmi_val2=[]; nmi_val3=[];

for epoch in range(epochs):
    model.train()
    B_tensor = Variable(B_tensor).cuda()
```

```
    H1, H2, H3, M = model(B_tensor) # H denotes low-embedding vectors, M is the reconstruction
        matrix
    # For pairwise_loss, we only need to compute those which have labels.
#     Our goal is to learn the low embedding representations of all vertices via a small portion of
     vertices with labels
    pairwise1 = torch.mm( torch.mm(H1[idx_train,:].t(), L), H1[idx_train,:] )# H'*L*H
    pairwise2 = torch.mm( torch.mm(H2[idx_train,:].t(), L), H2[idx_train,:] )
    pairwise3 = torch.mm( torch.mm(H3[idx_train,:].t(), L), H3[idx_train,:] )
    pairwise_loss = torch.trace(pairwise1) + torch.trace(pairwise2) + torch.trace(pairwise3)#
        trace(H'*L*H)
#     pairwise_loss = torch.trace(pairwise3)
    AE_loss = criterion_AE(M[0], B_tensor) + criterion_AE(M[1], H1) + criterion_AE(M[2], H2)#
        compute the reconstruction loss of AE
    total_loss = criterion(AE_loss, pairwise_loss) # total loss= recon_loss + lamd*pairwise_loss
    totalloss_list.append(total_loss.item()); pairwise_list.append(pairwise_loss.item());
        AE_list.append(AE_loss.item())
    if epoch%10 == 0:
        model.eval()
        H1, H2, H3, M = model(B_tensor)
        H1, H2, H3 = H1.cpu(), H2.cpu(), H3.cpu()
        nmi_train_H1 = cal_nmi(data=H1[idx_train.numpy(),:].detach().numpy(),
            labels=labels[idx_train.numpy()], n_clusters=7)
        nmi_train_H2 = cal_nmi(data=H2[idx_train.numpy(),:].detach().numpy(),
            labels=labels[idx_train.numpy()], n_clusters=7)
        nmi_train_H3 = cal_nmi(data=H3[idx_train.numpy(),:].detach().numpy(),
            labels=labels[idx_train.numpy()], n_clusters=7)
        nmi_val_H1 = cal_nmi(data=H1[idx_val.numpy(),:].detach().numpy(),
            labels=labels[idx_val.numpy()], n_clusters=7)
        nmi_val_H2 = cal_nmi(data=H2[idx_val.numpy(),:].detach().numpy(),
            labels=labels[idx_val.numpy()], n_clusters=7)
        nmi_val_H3 = cal_nmi(data=H3[idx_val.numpy(),:].detach().numpy(),
            labels=labels[idx_val.numpy()], n_clusters=7)
        print('epoch{}:total_loss:{:.6f}|AE_loss: {:.6f}|pair_loss: {:.6f}|NMI train(H1 H2 H3): '
              '{:.2f} {:.2f} {:.2f}|val: {:.2f} {:.2f} {:.2f}'.format
              (epoch, total_loss, AE_loss, pairwise_loss, nmi_train_H1, nmi_train_H2, nmi_train_H3,
                  nmi_val_H1,
               nmi_val_H2, nmi_val_H3))
        model.train()
        nmi_train1.append(nmi_train_H1); nmi_train2.append(nmi_train_H2);
            nmi_train3.append(nmi_train_H3)
        nmi_val1.append(nmi_val_H1); nmi_val2.append(nmi_val_H2); nmi_val3.append(nmi_val_H3)

    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt
host = host_subplot(111)
par = host.twinx()
host.set_xlabel("Epochs")
host.set_ylabel("Total-loss")
par.set_ylabel("Pairwise-loss")
p1, = host.plot(range(epochs), totalloss_list, 'r-', lw=2, label="Total-loss")
p2, = host.plot(range(epochs), AE_list, 'k-', lw=2, label="AE-loss")
p3, = par.plot(range(epochs), np.array(pairwise_list), 'b-', lw=2, label="Pairwise-loss")
leg = plt.legend(loc='best')
host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())
```

```
par.yaxis.get_label().set_color(p3.get_color())
leg.texts[2].set_color(p3.get_color())
plt.show()

plt.plot( nmi_train1, 'r-', lw=2, label='nmi_train1')
plt.plot( nmi_train2, 'y-', lw=2, label='nmi_train2')
plt.plot( nmi_train3, 'g-', lw=2, label='nmi_train3')
plt.plot( nmi_val1, 'r--', lw=2, label='nmi_val1')
plt.plot( nmi_val2, 'y--', lw=2, label='nmi_val2')
plt.plot( nmi_val3, 'g--', lw=2, label='nmi_val3')
# plt.plot(range(epochs), np.array(pairwise_list)*lamd, 'b-', lw=3, label='pairwise_loss')
plt.legend(loc='upper left');
plt.show()
```

GCN + pairwise loss

```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
import scipy.sparse as sp
from torch.autograd import Variable

def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    return labels_onehot

def
     load_data(path="C:/Users/kevin/Downloads/Coursework/GRAPH_ANALYTICS/pygcn/pygcn-master/data/cora/",
     dataset="cora"):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))

    idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                        dtype=np.dtype(str))
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    edges_ordered_flatten = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32)
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

```
    adj_unnormalize = adj.copy() # copy the unnormalized adjacency matrix

    features = normalize(features)
    adj = normalize(adj + sp.eye(adj.shape[0]))

    idx_train = range(270)
    idx_val = range(270, 2707)
    idx_test = range(2270, 2707)

    features = torch.FloatTensor(np.array(features.todense()))
    labels = torch.LongTensor(np.where(labels)[1])
    adj = sparse_mx_to_torch_sparse_tensor(adj)

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)
    idx_test = torch.LongTensor(idx_test)

    return adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx

def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
    return correct / len(labels)

def sparse_mx_to_torch_sparse_tensor(sparse_mx):
    """Convert a scipy sparse matrix to a torch sparse tensor."""
    sparse_mx = sparse_mx.tocoo().astype(np.float32)
    indices = torch.from_numpy(
        np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))
    values = torch.from_numpy(sparse_mx.data)
    shape = torch.Size(sparse_mx.shape)
    return torch.sparse.FloatTensor(indices, values, shape)

# Load data
# I use Kipf's load_data() and modified it by returning an unnormalized adjacency matrix
adj, features, labels, idx_train, idx_val, idx_test, adj_unnormalize = load_data('')
# adj_array = adj_unnormalize.toarray() # transform csr to dense array

# This loss function adds the AutoEncoder loss and the pairwise loss together.
# lamd is a tradeoff hyperparameter.
class Semi_Loss(torch.nn.Module):

    def __init__(self, lamd=0.01):
        super(Semi_Loss,self).__init__()
        self.lamd = lamd

    def forward(self, AE_loss, semi_loss):
        semi_loss = AE_loss + self.lamd * semi_loss
        return semi_loss
```

```
# build Laplace matrix. For this part, please refer to Section 4 of [1]
# L = D - O
labels_train = encode_onehot(labels[idx_train].numpy())
n = idx_train.numpy().size
O = labels_train.dot(labels_train.T) - np.eye(n) # O denotes pairwise constraint matrix
L = np.diag(O.sum(1)) - O # L = D - O, D is a diagonal matrix whose entries are row summation of O
L = torch.from_numpy(L).float() # convert numpy array into torch float tensor

from sklearn.cluster import KMeans
from sklearn.metrics.cluster import normalized_mutual_info_score
# calculate normalized mutual info to measure the clustering performance
def cal_nmi(data, labels, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++').fit(data)
    pred_tr = kmeans.labels_
    nmi = normalized_mutual_info_score(labels, pred_tr, average_method='arithmetic')
    return nmi

import math
import torch

from torch.nn.parameter import Parameter
from torch.nn.modules.module import Module

class GraphConvolution(Module):
    """
    Simple GCN layer, similar to https://arxiv.org/abs/1609.02907
    """

    def __init__(self, in_features, out_features, bias=True):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.FloatTensor(in_features, out_features))
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input, adj):
        support = torch.mm(input, self.weight)
        output = torch.spmm(adj, support)
        if self.bias is not None:
            return output + self.bias
        else:
            return output

    def __repr__(self):
        return self.__class__.__name__ + ' (' \
               + str(self.in_features) + ' -> ' \
               + str(self.out_features) + ')'
```

```
import torch.nn as nn
import torch.nn.functional as F
# from pygcn.layers import GraphConvolution

class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):# nhid2,
        super(GCN, self).__init__()

        self.encoder_gc1 = GraphConvolution(nfeat, nhid)
        self.encoder_gc2 = GraphConvolution(nhid, nclass)
        self.decoder_gc2 = GraphConvolution(nclass, nhid)
        self.decoder_gc1 = GraphConvolution(nhid, nfeat)
        self.dropout = dropout

    def forward(self, x, adj):
        x = F.relu(self.encoder_gc1(x, adj))
        x = F.dropout(x, self.dropout, training=self.training)
        h = self.encoder_gc2(x, adj)
        h = F.relu(h)
        d = F.relu(self.decoder_gc2(h, adj))
        x = F.dropout(x, self.dropout, training=self.training)
        d = self.decoder_gc1(d, adj)
        return d, h #F.log_softmax(x, dim=1)

features = features.cuda()
adj = adj.cuda()
L = L.cuda()
labels = labels.cpu()

Model and optimizer
model = GCN(nfeat=features.shape[1],
           nhid=512,
           nclass=128,#labels.max().item() + 1,
           dropout=0.5)
lamd = 4*10**-9
model = model.cuda()
optimizer = optim.Adam(model.parameters(),
                       lr=0.0004, weight_decay=0)
criterion = Semi_Loss(lamd)
criterion_AE = nn.MSELoss()
epochs = 1000
totalloss_list=[];pairwise_list=[];AE_list=[];nmi_train=[];nmi_val=[]

for epoch in range(epochs):
# def train(epoch):
#     t = time.time()
    model.train()
    optimizer.zero_grad()
    M,H = model(features, adj)
    pairwise = torch.mm( torch.mm(H[idx_train,:].t(), L), H[idx_train,:] ) # H'*L*H
    pairwise_loss = torch.trace(pairwise) # trace(H'*L*H)
    AE_loss = criterion_AE(M, features) # compute the reconstruction loss of AE
    total_loss = criterion(AE_loss, pairwise_loss) # total loss= recon_loss + lamd*pairwise_loss
    totalloss_list.append(total_loss.item()); pairwise_list.append(pairwise_loss.item());
        AE_list.append(AE_loss.item())
    if epoch%10 == 0:
        model.eval()
        M,H = model(features, adj)
        H = H.data.cpu()
```

```
        nmi_train_H = cal_nmi(data=H[idx_train.numpy(),:].data.cpu().numpy(),
            labels=labels[idx_train.numpy()], n_clusters=7)
        nmi_val_H = cal_nmi(data=H[idx_val.numpy(),:].detach().numpy(),
            labels=labels[idx_val.numpy()], n_clusters=7)
        print('epoch {}: total_loss:{:.6f}|AE_loss: {:.6f}|pair_loss: {:.6f}|NMI train: {:.2f}|val:
            {:.2f}'.format
              (epoch, total_loss, AE_loss, pairwise_loss, nmi_train_H, nmi_val_H))
        model.train()
        nmi_train.append(nmi_train_H); nmi_val.append(nmi_val_H)

#     optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt
host = host_subplot(111)
par = host.twinx()
host.set_xlabel("Epochs")
host.set_ylabel("Total-loss")
par.set_ylabel("Pairwise-loss")
p1, = host.plot(range(epochs), totalloss_list, 'r-', lw=2, label="Total-loss")
p2, = host.plot(range(epochs), AE_list, 'k-', lw=2, label="AE-loss")
p3, = par.plot(range(epochs), np.array(pairwise_list), 'b-', lw=2, label="Pairwise-loss")
leg = plt.legend(loc='best')
host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())
par.yaxis.get_label().set_color(p3.get_color())
leg.texts[2].set_color(p3.get_color())
plt.show()

plt.plot( nmi_train, 'r-', lw=2, label='nmi_train')
plt.plot( nmi_val, 'k-', lw=2, label='nmi_val')
# plt.plot(range(epochs), np.array(pairwise_list)*lamd, 'b-', lw=3, label='pairwise_loss')
plt.legend(loc='upper left');
plt.show()
print(np.max(nmi_val))
```

GCN + Cross-Entropy loss

```
from __future__ import print_function
from __future__ import division
import time
import argparse
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import scipy.sparse as sp

def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    return labels_onehot
```

```
def
    load_data(path="C:/Users/kevin/Downloads/Coursework/GRAPH_ANALYTICS/pygcn/pygcn-master/data/cora/",
    dataset="cora"):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))

    idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                        dtype=np.dtype(str))
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    edges_ordered_flatten = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                    dtype=np.int32)
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                    dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
    adj1 = adj.copy()

    features = normalize(features)
    adj = normalize(adj + sp.eye(adj.shape[0]))

    idx_train = range(270)
    idx_val = range(270, 2707)
    idx_test = range(2270, 2707)

    features = torch.FloatTensor(np.array(features.todense()))
    labels = torch.LongTensor(np.where(labels)[1])
    adj = sparse_mx_to_torch_sparse_tensor(adj)

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)
    idx_test = torch.LongTensor(idx_test)

    return adj, features, labels, idx_train, idx_val, idx_test, adj1

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    print('rowsum shape', rowsum.shape)
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx

def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
```

```
    return correct / len(labels)

def sparse_mx_to_torch_sparse_tensor(sparse_mx):
    """Convert a scipy sparse matrix to a torch sparse tensor."""
    sparse_mx = sparse_mx.tocoo().astype(np.float32)
    indices = torch.from_numpy(
        np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))
    values = torch.from_numpy(sparse_mx.data)
    shape = torch.Size(sparse_mx.shape)
    return torch.sparse.FloatTensor(indices, values, shape)

# Load data
adj, features, labels, idx_train, idx_val, idx_test, adj1 = load_data('')

import math
import torch

from torch.nn.parameter import Parameter
from torch.nn.modules.module import Module

class GraphConvolution(Module):
    """
    Simple GCN layer, similar to https://arxiv.org/abs/1609.02907
    """

    def __init__(self, in_features, out_features, bias=True):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.FloatTensor(in_features, out_features))
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input, adj):
        support = torch.mm(input, self.weight)
        output = torch.spmm(adj, support)
        if self.bias is not None:
            return output + self.bias
        else:
            return output

    def __repr__(self):
        return self.__class__.__name__ + ' (' \
               + str(self.in_features) + ' -> ' \
               + str(self.out_features) + ')'

import torch.nn as nn
import torch.nn.functional as F

class GCN(nn.Module):
```

```
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.encoder_gc1 = GraphConvolution(nfeat, nhid)
        self.encoder_gc2 = GraphConvolution(nhid, nclass)
        self.decoder_gc2 = GraphConvolution(nclass, nhid)
        self.decoder_gc1 = GraphConvolution(nhid, nfeat)
        self.dropout = dropout

    def forward(self, x, adj):
        x = F.relu(self.encoder_gc1(x, adj))
        x = F.dropout(x, self.dropout, training=self.training)
        h = self.encoder_gc2(x, adj)
        h = F.relu(h)
        x = F.relu(self.decoder_gc2(h, adj))
        x = self.decoder_gc1(x, adj)
        return F.log_softmax(x, dim=1), h, x

from sklearn.cluster import KMeans
from sklearn.metrics.cluster import normalized_mutual_info_score
# calculate normalized mutual info to measure the clustering performance
def cal_nmi(pred_tr, labels, n_clusters):
    nmi = normalized_mutual_info_score(labels, pred_tr, average_method='arithmetic')
    return nmi
def cal_nmi_data(data, labels, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++').fit(data)
    pred_tr = kmeans.labels_
    nmi = normalized_mutual_info_score(labels, pred_tr, average_method='arithmetic')
    return nmi

# Model and optimizer
model = GCN(nfeat=features.shape[1],
            nhid=32,
            nclass=labels.max().item() + 1,
            dropout=0.5)
optimizer = optim.Adam(model.parameters(),
                       lr=0.01, weight_decay=5e-4)
criterion_AE = nn.MSELoss()

def train(epoch):
#     t = time.time()
    model.train()
    optimizer.zero_grad()
    output,h,x_recon = model(features, adj)# h denotes latent representations
    preds = output.max(1)[1].type_as(labels)
    loss_recon = criterion_AE(features, x_recon)
    loss_semi = F.nll_loss(output[idx_train], labels[idx_train])
    loss_train = loss_recon + loss_semi
    acc_train = accuracy(output[idx_train], labels[idx_train])
    nmi_tr = cal_nmi(preds[idx_train].data.numpy(), labels[idx_train].data.numpy(), n_clusters=7)
    nmi_tr_h = cal_nmi_data(h[idx_train].data.numpy(), labels[idx_train].data.numpy(),
        n_clusters=7)
    loss_train.backward()
    optimizer.step()


    # Evaluate validation set performance separately,
    # deactivates dropout during validation run.
    model.eval()
```

```
    output,h,x_recon = model(features, adj)
    preds = output.max(1)[1].type_as(labels)

    loss_val = F.nll_loss(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])
    nmi_val = cal_nmi(preds[idx_val].data.numpy(), labels[idx_val].data.numpy(), n_clusters=7)
    nmi_val_h = cal_nmi_data(h[idx_val].data.numpy(), labels[idx_val].data.numpy(), n_clusters=7)
    print('Epoch: {:04d}'.format(epoch+1),
#          'loss_train: {:.4f}'.format(loss_train.item()),
          'acc_train: {:.4f}'.format(acc_train.item()),
          'nmi_train: {:.4f}'.format(nmi_tr),
          'nmi_train_h: {:.4f}'.format(nmi_tr_h),
#           'loss_val: {:.4f}'.format(loss_val.item()),
          'acc_val: {:.4f}'.format(acc_val.item()),
          'nmi_val: {:.4f}'.format(nmi_val),
          'nmi_val_h: {:.4f}'.format(nmi_val_h),
#           'time: {:.4f}s'.format(time.time() - t),
         )
    return loss_recon, loss_semi, loss_train, nmi_tr, nmi_tr_h, nmi_val, nmi_val_h


def test():
    model.eval()
    output,h,x_recon = model(features, adj)
    preds = output.max(1)[1].type_as(labels)
    loss_test = F.nll_loss(output[idx_test], labels[idx_test])
    acc_test = accuracy(output[idx_test], labels[idx_test])
    nmi_test = cal_nmi(preds[idx_test], labels[idx_test], n_clusters=7)
    print("Test set results:",
          "loss= {:.4f}".format(loss_test.item()),
          "nmi= {:.4f}".format(nmi_test),
          "accuracy= {:.4f}".format(acc_test.item()))


# Train model
# t_total = time.time()
epochs = 800
totalloss_list=[];loss_semi_list=[];AE_list=[];nmi_train=[];nmi_train_h=[];nmi_val=[];nmi_val_h=[]
for epoch in range(epochs):
    loss_recon, loss_semi, loss_train, nmi_tr, nmi_tr_h, nmi_v, nmi_v_h = train(epoch)
    totalloss_list.append(loss_train.item()); loss_semi_list.append(loss_semi.item());
        AE_list.append(loss_recon.item())
    nmi_train.append(nmi_tr); nmi_train_h.append(nmi_tr_h); nmi_val.append(nmi_v);
        nmi_val_h.append(nmi_v_h)
print("Optimization Finished!")
# print("Total time elapsed: {:.4f}s".format(time.time() - t_total))

# Testing
test()
# Plotting loss
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt
host = host_subplot(111)
par = host.twinx()
host.set_xlabel("Epochs")
host.set_ylabel("AE-loss")
par.set_ylabel("Cross-Entropy-loss")
p1, = host.plot(range(epochs), totalloss_list, 'r-', lw=2, label="Total-loss")
p2, = host.plot(range(epochs), AE_list, 'k-', lw=2, label="AE-loss")
```

```
p3, = par.plot(range(epochs), np.array(loss_semi_list), 'b-', lw=2, label="Cross-Entropy-loss")
leg = plt.legend(loc='best')
host.yaxis.get_label().set_color(p2.get_color())
leg.texts[0].set_color(p1.get_color())
par.yaxis.get_label().set_color(p3.get_color())
leg.texts[2].set_color(p3.get_color())
plt.show()

# Plotting nmi
plt.plot( nmi_train, 'r-', lw=2, label='nmi_train')
plt.plot( nmi_train_h, 'b-', lw=2, label='nmi_train_h')
plt.plot( nmi_val, 'r--', lw=2, label='nmi_val')
plt.plot( nmi_val_h, 'b--', lw=2, label='nmi_val_h')
plt.legend(loc='upper left');
plt.show()
print("nmi_val: {:.4f}".format(np.max(nmi_val)), "nmi_val_h: {:.4f}".format(np.max(nmi_val_h)))
```