Accuracy = 0.8340

**Train.py**

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import time

import argparse

import numpy as np

import torch

import torch.nn.functional as F

import torch.optim as optim

from utils import load\_data, accuracy

from models import GCN

# Training settings

parser = argparse.ArgumentParser()

parser.add\_argument('--no-cuda', action='store\_true', default=False,

help='Disables CUDA training.')

parser.add\_argument('--fastmode', action='store\_true', default=False,

help='Validate during training pass.')

parser.add\_argument('--seed', type=int, default=42, help='Random seed.')

parser.add\_argument('--epochs', type=int, default=200,

help='Number of epochs to train.')

parser.add\_argument('--lr', type=float, default=0.01,

help='Initial learning rate.')

parser.add\_argument('--weight\_decay', type=float, default=5e-4,

help='Weight decay (L2 loss on parameters).')

parser.add\_argument('--hidden', type=int, default=16,

help='Number of hidden units.')

parser.add\_argument('--dropout', type=float, default=0.5,

help='Dropout rate (1 - keep probability).')

args = parser.parse\_args()

args.cuda = not args.no\_cuda and torch.cuda.is\_available()

np.random.seed(args.seed)

torch.manual\_seed(args.seed)

if args.cuda:

torch.cuda.manual\_seed(args.seed)

# Load data

adj, features, labels, idx\_train, idx\_val, idx\_test = load\_data()

# Model and optimizer

model = GCN(nfeat=features.shape[1],

nhid=args.hidden,

nclass=labels.max().item() + 1,

dropout=args.dropout)

optimizer = optim.Adam(model.parameters(),

lr=args.lr, weight\_decay=args.weight\_decay)

if args.cuda:

model.cuda()

features = features.cuda()

adj = adj.cuda()

labels = labels.cuda()

idx\_train = idx\_train.cuda()

idx\_val = idx\_val.cuda()

idx\_test = idx\_test.cuda()

def train(epoch):

t = time.time()

model.train()

optimizer.zero\_grad()

output = model(features, adj)

loss\_train = F.nll\_loss(output[idx\_train], labels[idx\_train])

acc\_train = accuracy(output[idx\_train], labels[idx\_train])

loss\_train.backward()

optimizer.step()

if not args.fastmode:

# Evaluate validation set performance separately,

# deactivates dropout during validation run.

model.eval()

output = model(features, adj)

loss\_val = F.nll\_loss(output[idx\_val], labels[idx\_val])

acc\_val = accuracy(output[idx\_val], labels[idx\_val])

print('Epoch: {:04d}'.format(epoch+1),

'loss\_train: {:.4f}'.format(loss\_train.item()),

'acc\_train: {:.4f}'.format(acc\_train.item()),

'loss\_val: {:.4f}'.format(loss\_val.item()),

'acc\_val: {:.4f}'.format(acc\_val.item()),

'time: {:.4f}s'.format(time.time() - t))

def test():

model.eval()

output = model(features, adj)

loss\_test = F.nll\_loss(output[idx\_test], labels[idx\_test])

acc\_test = accuracy(output[idx\_test], labels[idx\_test])

print("Test set results:",

"loss= {:.4f}".format(loss\_test.item()),

"accuracy= {:.4f}".format(acc\_test.item()))

# Train model

t\_total = time.time()

for epoch in range(args.epochs):

train(epoch)

print("Optimization Finished!")

print("Total time elapsed: {:.4f}s".format(time.time() - t\_total))

# Testing

test()

**layers.py**

import math

import torch

from torch.nn.parameter import Parameter

from torch.nn.modules.module import Module

class GCNLayer(Module):

def \_\_init\_\_(self, input\_features, output\_features, bias=True):

super(GCNLayer, self).\_\_init\_\_()

self.input\_features = input\_features

self.output\_features = output\_features

self.weight = Parameter(torch.FloatTensor(input\_features, output\_features))

if bias:

self.bias = Parameter(torch.FloatTensor(output\_features))

else:

self.register\_parameter('bias', None)

self.reset\_parameters()

def forward(self, input, adj): # input is H, adj is (D^-1/2)(A)(D^-1/2)

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 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)

**models.py**

import torch.nn as nn

import torch.nn.functional as F

from layers import GCNLayer

class GCN(nn.Module):

def \_\_init\_\_(self, nfeat, nhid, nclass, dropout):

super(GCN, self).\_\_init\_\_()

self.gc1 = GCNLayer(nfeat, nhid)

self.gc2 = GCNLayer(nhid, nclass)

self.dropout = dropout

def forward(self, x, adj):

x = F.relu(self.gc1(x, adj))

x = F.dropout(x, self.dropout, training=self.training)

x = self.gc2(x, adj)

return F.log\_softmax(x, dim=1)

**utils.py**

import numpy as np

import scipy.sparse as sp

import torch

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="../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 = 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)

features = normalize(features)

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

idx\_train = range(140)

idx\_val = range(200, 500)

idx\_test = range(500, 1500)

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

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)