GCN代码解析 of GitHub：Semi-supervised

本文解析的代码是论文Semi-Supervised Classification with Graph Convolutional Networks作者提供的实现代码。

原GitHub：Graph Convolutional Networks in PyTorch

本人增加结果可视化（使用 t-SNE 算法）的GitHub：Result-Visualization-of-Graph-Convolutional-Networks-in-PyTorch。本文作代码解析的也是这一个。

文章目录

- train.py
  - 函数定义
    - 版本兼容
    - 路径初始化
    - 所需要的函数库
    - 显示超参数函数：show_Hyperparameter(args)
    - 训练函数：train(epoch)
      - 训练初始化
      - 计算模型输出结果
      - 计算训练集损失
      - 反向传播及优化
      - 计算验证集结果
      - 打印训练集和验证集的结果信息
    - 测试函数：test()

        - 降维函数：t_SNE(output, dimention)
        - Visdom可视化函数：
    - 主函数代码
        - 解析命令行
        - 显示超参数
        - 设定随机数种子
        - 读取数据
        - 定义模型
        - 定义优化器
        - CPU转CUDA
        - 训练模型
        - 测试模型
        - 计算预测值
        - 结果可视化
- utils.py
    - 标签编码函数：encode_onehot(labels)
    - 特征归一化函数：normalize(mx)
    - 数据集读取函数：load_data(path, dataset)
        - labels 预处理
        - feature 预处理
        - 构建邻接矩阵 adj
- 在这里插入图片描述
        - 数据集划分

- 数据类型转为tensor
- 返回数据
- 计算准确率函数：accuracy(output, labels)
- 稀疏矩阵转稀疏张量函数：sparse_mx_to_torch_sparse_tensor(sparse_mx)
- layers.py
- models.py
- 2020.03.12修改，使用有向图邻接矩阵：

---

train.py

函数定义

版本兼容

```
from __future__ import division
from __future__ import print_function
```

- 第一条语句：

在 Python2 中导入未来的支持的语言特征中division (精确除法)，即from __future__ import division ，当我们在程序中没有导入该特征时，"/“操作符执行的只能是整除，也就是取整数，只有当我们导入division(精确算法)以后，”/"执行的才是精确算法。

- 第二条语句：

在开头加上from __future__ import print_function这句之后，即使在python2.X，使用print就得像python3.X那样加括号使用。python2.X中print不需要括号，而在python3.X中则需要。

路径初始化

```
# 路径初始化
import os, sys
curPath = os.path.abspath(os.path.dirname(__file__))
rootPath = os.path.split(curPath)[0]
sys.path.append(rootPath)
sys.path.append('E:\\Anaconda\\lib\\site-packages\\')
# print(sys.path)
print('Path initialization finished！\n')
```

这部分是函数库的路径初始化。本人使用的环境是Anaconda下的Python环境，需要的函数库都安装在该路径下。而命令行运行程序时，程序对于函数库的搜索路径是Python的环境，二者并不相同。

要在命令行运行程序时使用Anaconda下的环境，就需要通过上面的代码，经Anaconda路径下的函数库增加到搜索路径。

更多的关于原生Python环境和Anaconda环境的细节，参见另一篇博客：

[python+pip] 使用pip将函数库安装到Python环境或Anaconda环境

所需要的函数库

```
# 可视化增加路径
from time import time
from sklearn import manifold, datasets


# visdom显示模块
from visdom import Visdom

import time
import argparse
import numpy as np

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

from pygcn.utils import load_data, accuracy
from pygcn.models import GCN
```

显示超参数函数：show_Hyperparameter(args)

```
def show_Hyperparameter(args):
    argsDict = args.__dict__
    print(argsDict)
    print('the settings are as following:')
    for key in argsDict:
        print(key,':',argsDict[key])
```

将命名空间 namespace 的超参数，通过其带有的`args.__dict__`方法，转化为字典类。最后通过遍历字典打印超参数。

默认的超参数是这样的：

```
the settings are as following:
no_cuda : False
```

fastmode : False
seed : 42
epochs : 200
lr : 0.01
weight_decay : 0.0005
hidden : 16
dropout : 0.5

训练函数：train(epoch)

```
def train(epoch):
    t = time.time()
    '''将模型转为训练模式，并将优化器梯度置零'''
    model.train()
    optimizer.zero_grad()
    '''计算输出时，对所有的节点计算输出'''
    output = model(features, adj)
    '''损失函数，仅对训练集节点计算，即：优化仅对训练集数据进行'''
    loss_train = F.nll_loss(output[idx_train], labels[idx_train])
    # 计算准确率
    acc_train = accuracy(output[idx_train], labels[idx_train])
    # 反向传播
    loss_train.backward()
    # 优化
    optimizer.step()

    '''fastmode ? '''
    if not args.fastmode:
        # Evaluate validation set performance separately,
        # deactivates dropout during validation run.
        model.eval()
        output = model(features, adj)

    '''验证集 loss 和 accuracy '''
    loss_val = F.nll_loss(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])
    '''输出训练集+验证集的 loss 和 accuracy '''
    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))
```

传入参数epoch为：当前训练的epoch数。

训练函数在循环中被调用，train()函数本身没有循环（即train()函数表示一次循环的步骤）。

训练初始化

```
    '''将模型转为训练模式，并将优化器梯度置零'''
model.train()
optimizer.zero_grad()
```

计算模型输出结果

```
    '''计算输出时，对所有的节点计算输出'''
output = model(features, adj)
```

计算model输出，总是对全部样本进行。在计算损失和反向传播时，才对训练集、验证集、测试集进行分别的操作。

计算训练集损失

```
    '''损失函数，仅对训练集节点计算，即：优化仅对训练集数据进行'''
loss_train = F.nll_loss(output[idx_train], labels[idx_train])
# 计算准确率
acc_train = accuracy(output[idx_train], labels[idx_train])
```

半监督的代码实现，是仅仅对训练集（即：我们认为标签已知的子集）计算损失函数。

反向传播及优化

```
    # 反向传播
loss_train.backward()
# 优化
optimizer.step()
```

反向传播对训练集数据进行（即：我们认为标签已知的子集）。

通过计算训练集损失和反向传播及优化，带标签的 label 信息就可以 smooth 到整个图上（label information is smoothed over the graph）。

计算验证集结果

```
    '''验证集 loss 和 accuracy '''
loss_val = F.nll_loss(output[idx_val], labels[idx_val])
acc_val = accuracy(output[idx_val], labels[idx_val])
```

打印训练集和验证集的结果信息

```
    '''输出训练集+验证集的 loss 和 accuracy '''
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))
```

测试函数：test()

```
def test():
    model.eval()    # model转为测试模式
    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()))
    return output      # 可视化返回output
```

先是通过`model.eval()`转为测试模式，之后计算输出，并单独对测试集计算损失函数和准确率。

降维函数：t_SNE(output, dimention)

```
# t-SNE 降维
def t_SNE(output, dimention):
    # output:待降维的数据
    # dimention：降低到的维度
    tsne = manifold.TSNE(n_components=dimention, init='pca', random_state=0)
    result = tsne.fit_transform(output)
    return result
```

传入变量为

- `output`：待降维的数据
- `dimention`：降低到的维度

输入数据`output`是对模型测试得到的tensor类型的`output`转为ndarray类型，具体的信息为：

```
[[-5.3864675  -5.8370166  -5.6641455  ... -0.05461597 -4.686558
  -5.4951925 ]
 [-1.9110445  -3.6501741  -0.8442404  ... -3.3035958  -1.5382951
  -2.0365703 ]
 [-0.16186619 -3.470789   -3.589233   ... -3.9754026  -3.4787045
  -3.3947954 ]
 ...
 [-1.9097705  -0.5042465  -3.1999087  ... -3.0369117  -3.6273246
  -2.5524645 ]
 [-2.6523461  -2.9252334  -2.6154044  ... -3.0893543  -3.3290434
```

```
 -0.3563683 ]
 [-4.6700363  -4.532374   -4.5864363  ... -0.091573   -4.373736
  -3.9875987 ]]
<class 'numpy.ndarray'>
(2708, 7)
```

输出数据result的信息为（以 dimension=2 为例）：

```
[[ 64.82651     8.086376 ]
 [-51.36745   -23.28821  ]
 [-42.840324   34.0754   ]
 ...
 [ -2.6561208  54.75794  ]
 [-63.83643     1.8319011]
 [ 49.33189    17.178907 ]]
<class 'numpy.ndarray'>
(2708, 2)
```

Visdom可视化函数：

```
def Visualization(vis, result, labels,title):
    # vis: Visdom对象
    # result: 待显示的数据，这里为t_SNE()函数的输出
    # label: 待显示数据的标签
    # title: 标题
    vis.scatter(
        X = result,
        Y = labels+1,          # 将label的最小值从0变为1，显示时label不可为0
       opts=dict(markersize=3,title=title),
    )
```

传入参数为：

vis: Visdom对象
result: 待显示的数据，这里为t_SNE()函数的输出
label: 待显示数据的标签
title: 标题

输入数据result即为t_SNE()函数的输出，其信息为：

```
[[ 64.82651     8.086376 ]
 [-51.36745   -23.28821  ]
 [-42.840324   34.0754   ]
 ...
 [ -2.6561208  54.75794  ]
 [-63.83643     1.8319011]
 [ 49.33189    17.178907 ]]
<class 'numpy.ndarray'>
```

(2708, 2)

输入数据labels的信息为：

```
[4 2 0 ... 1 6 4]
<class 'numpy.ndarray'>
(2708,)
```

调用散点图函数scatter()作图。

至于为什么要使用Visodm作为可视化工具 ，原因是在命令行运行程序时，无法使用matplotlib库来进行可视化。原因很简单，自己试试就知道了。

关于Visdom的使用的细节，见另一篇博客：Visdom：Python可视化神器

主函数代码

解析命令行

```
# 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()
```

显示超参数

```
# 显示args
show_Hyperparameter(args)
```

详见 显示超参数函数：show_Hyperparameter(args)

设定随机数种子

```
# 设置随机数种子
np.random.seed(args.seed)
torch.manual_seed(args.seed)
if args.cuda:
    torch.cuda.manual_seed(args.seed)
```

`np.random.seed()`函数是为CPU运行设定种子，使得CPU运行时产生的随机数一样。

比如下面的示例：

```
# 随机数一样
random.seed(1)
print('随机数3：',random.random())
random.seed(1)
print('随机数4：',random.random())
random.seed(2)
print('随机数5：',random.random())

'''
随机数1： 0.7643602170615428
随机数2： 0.31630323818329664
随机数3： 0.13436424411240122
随机数4： 0.13436424411240122
随机数5： 0.9560342718892494
'''
```

`torch.manual_seed()`函数是为GPU运行设定种子，使得GPU运行时产生的随机数一样。

比如下面的 Demo：

```
torch.manual_seed(2)  #为CPU设置种子用于生成随机数，以使得结果是确定的
print(torch.rand(2))

if args.cuda:
    torch.cuda.manual_seed(args.seed) #为当前GPU设置随机种子;
    # 如果使用多个GPU，应该使用 torch.cuda.manual_seed_all()为所有的GPU设置种子。
```

-

通过设定随机数种子的好处是，使模型初始化的可学习参数相同，从而使每次的运行结果可以复现。

读取数据

```
# Load data
adj, features, labels, idx_train, idx_val, idx_test = load_data()        # 返回可视化要用的labels
```

详细的方法、返回值数据类型，见：[数据集读取函数：load_data(path, dataset)](#数据集读取函数：load_data(path, dataset))
读取的结果都是tensor类型的。其中各个变量：

- adj：是torch.sparse，已进行归一化
- features：归一化后的特征
- labels：int类型的标签，注意并不是onehot编码的形式。具体如下：

```
tensor([4, 2, 0,  ..., 1, 6, 4])
<class 'torch.Tensor'>
torch.Size([2708])
```

- idx_train、idx_val、idx_test：训练集、验证集、测试集中样本的序号。

定义模型

```
# Model
model = GCN(nfeat=features.shape[1],
            nhid=args.hidden,
            nclass=labels.max().item() + 1,    # 对Cora数据集，为7，即类别总数。
            dropout=args.dropout)
```

主要是设定nfeat、nhid、nclass和dropout这四个参数值。具体的model和layers的定义，见：layers.py 和 models.py。

定义优化器

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

选用Adam优化函数，学习率lr，weight_dacay由命令行指定。

CPU转CUDA

```
# to CUDA
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()
```

训练模型

```
# 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))
```

调用epochs次循环，其中 train(epoch) 是一次训练。

测试模型

```
# Testing
output=test()          # 返回output
```

得到的output的信息如下：

```
tensor([[-5.3865, -5.8370, -5.6641,  ..., -0.0546, -4.6866, -5.4952],
        [-1.9110, -3.6502, -0.8442,  ..., -3.3036, -1.5383, -2.0366],
        [-0.1619, -3.4708, -3.5892,  ..., -3.9754, -3.4787, -3.3948],
        ...,
        [-1.9098, -0.5042, -3.1999,  ..., -3.0369, -3.6273, -2.5525],
        [-2.6523, -2.9252, -2.6154,  ..., -3.0894, -3.3290, -0.3564],
        [-4.6700, -4.5324, -4.5864,  ..., -0.0916, -4.3737, -3.9876]],
       device='cuda:0', grad_fn=<LogSoftmaxBackward>)
<class 'torch.Tensor'>
torch.Size([2708, 7])
```

计算预测值

```
# 计算预测值
preds = output.max(1)[1].type_as(labels)
```

结果可视化

```
# output的格式转换
output=output.cpu().detach().numpy()
labels=labels.cpu().detach().numpy()
preds=preds.cpu().detach().numpy()

# Visualization with visdom
vis=Visdom(env='pyGCN Visualization')

# ground truth 可视化
result_all_2d=t_SNE(output,2)
Visualization(vis,result_all_2d,labels,
              title='[ground truth of all samples]\n Dimension reduction to %dD' %
(result_all_2d.shape[
result_all_3d=t_SNE(output,3)
```

```
Visualization(vis,result_all_3d,labels,
          title='[ground truth of all samples]\n Dimension reduction to %dD' %
(result_all_3d.shape[

# 预测结果可视化
result_test_2d=t_SNE(output[idx_test.cpu().detach().numpy()],2)
Visualization(vis,result_test_2d,preds[idx_test.cpu().detach().numpy()],
          title='[prediction of test set]\n Dimension reduction to %dD' %
(result_test_2d.shape[1]))
result_test_3d=t_SNE(output[idx_test.cpu().detach().numpy()],3)
Visualization(vis,result_test_3d,preds[idx_test.cpu().detach().numpy()],
          title='[prediction of test set]\n Dimension reduction to %dD' %
(result_test_3d.shape[1]))
```

需要注意的是，这里输入的labels是[load_data()](#数据集读取函数：load_data(path, dataset))返回值labels从tensor转为ndarray格式的结果，是int类型的标签，不是onehot编码的结果。labels的信息如下：

```
[4 2 0 ... 1 6 4]
<class 'numpy.ndarray'>
(2708,)
```

preds的格式于labels相同。

降维函数见t_SNE()：[降维函数：t_SNE(output, dimention)](#降维函数：t_SNE(output, dimention)) 。

可视化函数见Visualization()：Visdom可视化函数 。

---

utils.py

标签编码函数：encode_onehot(labels)

```
def encode_onehot(labels):
    classes = set(labels)       # set() 函数创建一个无序不重复元素集

    # enumerate()函数生成序列，带有索引i和值c。
    # 这一句将string类型的label变为int类型的label，建立映射关系
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    # map() 会根据提供的函数对指定序列做映射。
    # 这一句将string类型的label替换为int类型的label
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    # 返回int类型的label
    return labels_onehot
```

该函数用于改变标签labels的编码格式。将离散的字符串类型的labels，使用onehot编码，得到onehot编码形式的labels。

onehot编码，又称“独热编码”。其实就是用N位状态寄存器编码N个状态。每个状态都有独立的寄存器位，且这些寄存器位中只有一位有效，说白了就是只能有一个状态。

更多关于onehot编码的细节，参见博客：[数据预处理] onehot编码：是什么，为什么，怎么样。

对于Cora数据集的labels，处理前是离散的字符串标签：

[Genetic_Algorithms’， ‘Probabilistic_Methods’， ‘Reinforcement_Learning’，
‘Neural_Networks’， ‘Theory’， ‘Case_Based’， ‘Rule_Learning’]

对labels进行onehot编码后的结果如下：

```
# 'Genetic_Algorithms': array([1., 0., 0., 0., 0., 0., 0.]),
# 'Probabilistic_Methods': array([0., 1., 0., 0., 0., 0., 0.]),
# 'Reinforcement_Learning': array([0., 0., 1., 0., 0., 0., 0.]),
# 'Neural_Networks': array([0., 0., 0., 1., 0., 0., 0.]),
# 'Theory': array([0., 0., 0., 0., 1., 0., 0.]),
# 'Case_Based': array([0., 0., 0., 0., 0., 1., 0.]),
# 'Rule_Learning': array([0., 0., 0., 0., 0., 0., 1.])}
```

这里再附上使用onehot编码对Cora数据集的样本标签进行编码的Demo：

```
from pygcn.utils import encode_onehot
import numpy as np

'''labels的onehot编码，前后结果对比'''
# 读取原始数据集
path="C:/Users/73416/PycharmProjects/PyGCN_Visualization/data/cora/"
dataset = "cora"
idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                    dtype=np.dtype(str))

RawLabels=idx_features_labels[:, -1]
print("原始论文类别（label）： \n",RawLabels)
# ['Neural_Networks' 'Rule_Learning' 'Reinforcement_Learning' ...
# 'Genetic_Algorithms' 'Case_Based' 'Neural_Networks']
print(len(RawLabels))       # 2708

classes = set(RawLabels)       # set() 函数创建一个无序不重复元素集
print("原始标签的无序不重复元素集\n", classes)
# {'Genetic_Algorithms', 'Probabilistic_Methods', 'Reinforcement_Learning',
'Neural_Networks', 'Theory', 'Case_Based', 'Rule_Learning'}
```

```
# enumerate()函数生成序列，带有索引i和值c。
# 这一句将string类型的label变为onehot编码的label，建立映射关系
classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                enumerate(classes)}
print("原始标签与onehot编码结果的映射字典\n",classes_dict)
#  {'Genetic_Algorithms': array([1., 0., 0., 0., 0., 0., 0.]), 'Probabilistic_Methods':
array([0., 1., 0., 0., 0., 0., 0.]),
#   'Reinforcement_Learning': array([0., 0., 1., 0., 0., 0., 0.]), 'Neural_Networks':
array([0., 0., 0., 1., 0., 0., 0.]),
#   'Theory': array([0., 0., 0., 0., 1., 0., 0.]), 'Case_Based': array([0., 0., 0., 0., 0., 1., 0.]),
#   'Rule_Learning': array([0., 0., 0., 0., 0., 0., 1.])}

# map() 会根据提供的函数对指定序列做映射。
# 这一句将string类型的label替换为onehot编码的label
labels_onehot = np.array(list(map(classes_dict.get, RawLabels)),
                         dtype=np.int32)
print("onehot编码的论文类别（label）：\n",labels_onehot)
# [[0 0 0... 0 0 0]
#  [0 0 0... 1 0 0]
#  [0 1 0 ... 0 0 0]
#  ...
#  [0 0 0 ... 0 0 1]
#  [0 0 1 ... 0 0 0]
#  [0 0 0 ... 0 0 0]]
print(labels_onehot.shape)
# (2708, 7)
```

特征归一化函数：normalize(mx)

```
def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))                # (2708, 1)
    r_inv = np.power(rowsum, -1).flatten()      # (2708,)
    r_inv[np.isinf(r_inv)] = 0.                 # 处理除数为0导致的inf
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx
```

传入参数mx：传入的特征矩阵。对于Cora数据集来说，2708行每一行是一个样本，一个样本有1433列对应1433个特征。

传入的mx是稀疏矩阵的类型：

```
<2708x1433 sparse matrix of type '<class 'numpy.float32'>'
        with 49216 stored elements in Compressed Sparse Row format>
```

函数结果：归一化后的特征矩阵（即每一行元素求和为1），依旧是稀疏矩阵的类型。

```
<2708x1433 sparse matrix of type '<class 'numpy.float32'>'
	with 49216 stored elements in Compressed Sparse Row format>
```

实现方式：对mx每一行求和，取倒数之后的结果就是每一行非零元素（即1）归一化的数值，再与原mx作点乘（目的是将归一化数值替换掉原来的1，即将归一化数值与1相乘）。

以第一行（第一个样本）为例子：

```
sample1_label=RawFeature[0,:]
sumA=sample1_label.sum()
```

即第一行（第一个样本）有20个非零值。

第一行归一化的结果为：

```
  (0, 1236)	0.05
  (0, 1209)	0.05
  (0, 1205)	0.05
  (0, 902)	0.05
  (0, 845)	0.05
  (0, 734)	0.05
  (0, 702)	0.05
  (0, 698)	0.05
  (0, 648)	0.05
  (0, 619)	0.05
  (0, 521)	0.05
  (0, 507)	0.05
  (0, 456)	0.05
  (0, 351)	0.05
  (0, 252)	0.05
  (0, 176)	0.05
  (0, 125)	0.05
  (0, 118)	0.05
```


归一化后的值正好是 1/20=0.051/20=0.05。

测试该函数的Demo放到下面了：

```
import numpy as np
import scipy.sparse as sp
from pygcn.utils import normalize

'''测试归一化函数'''
# 读取原始数据集
path="C:/Users/73416/PycharmProjects/PyGCN_Visualization/data/cora/"
dataset = "cora"
idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
```

```
                                    dtype=np.dtype(str))

RawFeature = idx_features_labels[:, 1:-1]
RawFeature=RawFeature.astype(int)
sample1_label=RawFeature[0,:]
sumA=sample1_label.sum()
print("原始的feature\n",RawFeature)
# type ndarray
# [['0' '0' '0'... '0' '0' '0']
#  ['0' '0' '0'... '0' '0' '0']
#  ['0' '0' '0'...'0' '0' '0']
#  ...
#  ['0' '0' '0'...'0' '0' '0']
# ['0' '0' '0'... '0' '0' '0']
# ['0' '0' '0'...'0' '0' '0']]
print(RawFeature.shape)
# (2708, 1433)

features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
# <2708x1433 sparse matrix of type '<class 'numpy.float32'>'
#  with 49216 stored elements in Compressed Sparse Row format>
print("csr_matrix之后的feature\n",features)
# type csr_matrix
# (0, 0)        0.0
# (0, 1)        0.0
# (0, 2)        0.0
# (0, 3)        0.0
# (0, 4)        0.0
# ::
# (2707, 1428)    0.0
# (2707, 1429)    0.0
# (2707, 1430)    0.0
# (2707, 1431)    0.0
# (2707, 1432)    0.0
print(features.shape)
# (2708, 1433)

# features = normalize(features)
rowsum = np.array(features.sum(1))      # (2708, 1)
r_inv = np.power(rowsum, -1).flatten()  # (2708,)
r_inv[np.isinf(r_inv)] = 0.             # 处理除数为0导致的inf
r_mat_inv = sp.diags(r_inv)
# <2708x2708 sparse matrix of type '<class 'numpy.float32'>'
#  with 2708 stored elements (1 diagonals) in DIAgonal format>
mx = r_mat_inv.dot(features)
print('normalization之后的feature\n',mx)
# (0, 176)    0.05
```

```
# (0, 125)    0.05
# (0, 118)    0.05
# ::
# (1, 1425)   0.05882353
# (1, 1389)   0.05882353
# (1, 1263)   0.05882353
# ::
# (2707, 136)   0.05263158
# (2707, 67)    0.05263158
# (2707, 19)    0.05263158
```

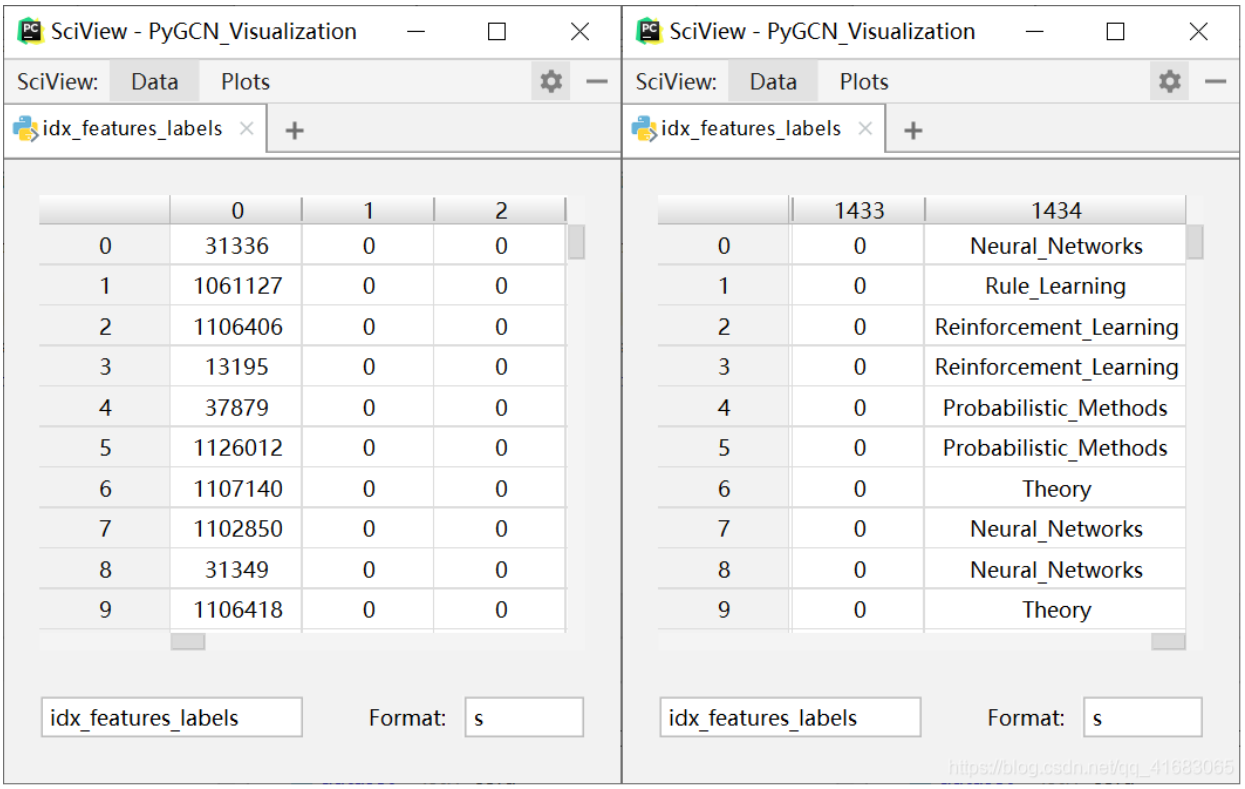
数据集读取函数：load_data(path, dataset)

```
'''数据读取'''
# 更改路径。由../改为C:\Users\73416\PycharmProjects\PyGCN
def
load_data(path="C:/Users/73416/PycharmProjects/PyGCN_Visualization/data/cora/",
 dataset="cora"):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))

    '''
    cora.content 介绍：
    cora.content共有2708行，每一行代表一个样本点，即一篇论文。
    每一行由三部分组成:
    是论文的编号，如31336；
    论文的词向量，一个有1433位的二进制；
    论文的类别，如Neural_Networks。总共7种类别（label）
    第一个是论文编号，最后一个是论文类别，中间是自己的信息（feature）
    '''

    '''读取feature和label'''
    # 以字符串形式读取数据集文件：各自的信息。
    idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                        dtype=np.dtype(str))

    # csr_matrix：Compressed Sparse Row marix，稀疏np.array的压缩
    # idx_features_labels[:, 1:-1]表明跳过论文编号和论文类别，只取自己的信息（feature
of node）
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)

    # idx_features_labels[:, -1]表示只取最后一个，即论文类别，得到的返回值为int类型的
label
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    # idx_features_labelsidx_features_labels[:, 0]表示取论文编号
```

```
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)

    # 通过建立论文序号的序列，得到论文序号的字典
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                    dtype=np.int32)
    # 进行一次论文序号的映射
    # 论文编号没有用，需要重新的其进行编号（从0开始），然后对原编号进行替换。
    # 所以目的是把离散的原始的编号，变成0 - 2707的连续编号
    edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                     dtype=np.int32).reshape(edges_unordered.shape)

    # coo_matrix()：系数矩阵的压缩。分别定义有那些非零元素，以及各个非零元素对应的row和col，最后定义稀疏矩阵的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)

    # feature和adj归一化
    features = normalize(features)
    adj = normalize(adj + sp.eye(adj.shape[0]))

    # train set, validation set, test set的分组。
    idx_train = range(140)
    idx_val = range(200, 500)
    idx_test = range(500, 1500)

    # 数据类型转tensor
    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)

    # 返回数据
    return adj, features, labels, idx_train, idx_val, idx_test
```

对于Cora数据集来说，读取到的第一手结果是这样的：

第一列：各个样本的标号（论文编号）
第二列-倒数第二列：各个样本的feature
第三列：各个样本的label



labels 预处理

即对 labels 进行 onehot 编码。

```
    # idx_features_labels[:, -1]表示只取最后一个，即论文类别，得到的返回值为int类型的label
    labels = encode_onehot(idx_features_labels[:, -1])
```

feature 预处理

即对 feature 进行归一化。由于 feature 为维度较大的稀疏矩阵，故使用 scipy.sparse 来处理。

```
    # csr_matrix：Compressed Sparse Row marix，稀疏np.array的压缩
    # idx_features_labels[:, 1:-1]表明跳过论文编号和论文类别，只取自己的信息（feature of node)
    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    ......
    # feature和adj归一化
    features = normalize(features)
```

关于处理方法，已经在 特征归一化函数：normalize(mx) 写明了。

构建邻接矩阵 adj

整体来说分两部分：

序号预处理：将非连续的离散序号，转化为连续的离散序号（0，1，2，.....，2707）。

根据预处理后的序号，将引用关系转化为邻接矩阵adj。

```
# idx_features_labelsidx_features_labels[:, 0]表示取论文编号
idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
```

这句是从idx_features_labels中取出idx，即样本的序号（与样本一一对应）。

```
# 通过建立论文序号的序列，得到论文序号的字典
idx_map = {j: i for i, j in enumerate(idx)}
```

这句是建立原有的样本序号到连续的离散序号（0，1，2，……，2707）之间的映射关系。
需要说明的是，样本的序号与样本一一对应，即不存在重复的情况，所以排序后的样本序号直接从0开始数就OK。

```
# 读取图的边（论文间的引用关系）
# cora.cites共5429行， 每一行有两个论文编号，表示第一个编号的论文先写，第二个编号的论文引用第一个编号的论文。
edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                dtype=np.int32)
```

以Cora数据集为例，这一句是从数据集文件中读取论文间的引用关系，即Graph中边。
读取的结果如下所示：

```
[[     35    1033]
 [     35  103482]
 [     35  103515]
 ...
 [ 853118 1140289]
 [ 853155  853118]
 [ 954315 1155073]]
```

需要注意，这里的样本序号还是原始的样本序号，还没映射到预处理后的序号。

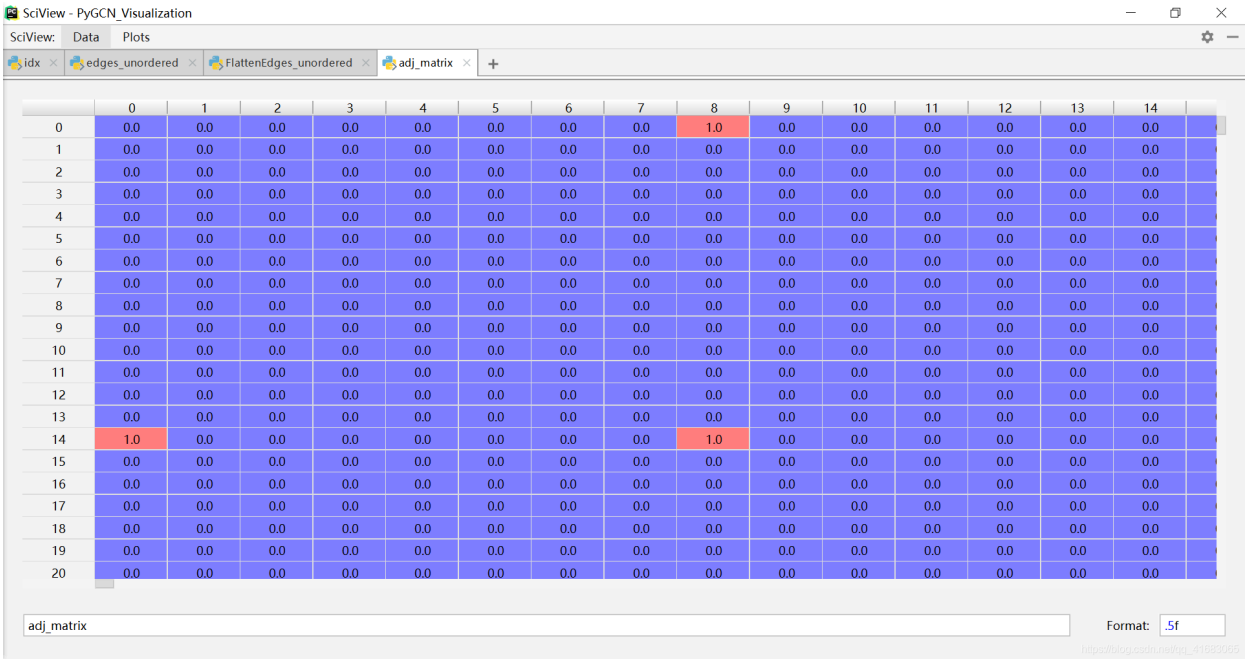
```
# 进行一次论文序号的映射
# 论文编号没有用，需要重新的其进行编号（从0开始），然后对原编号进行替换。
# 所以目的是把离散的原始的编号，变成0 - 2707的连续编号
edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                 dtype=np.int32).reshape(edges_unordered.shape)
```

这一句代码实现了样本序号的映射。用到的函数有map()，list()和reshape()。

```
# coo_matrix()：系数矩阵的压缩。分别定义有那些非零元素，以及各个非零元素对应的row和col，最后定义稀疏矩阵的shape。
adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                    shape=(2708, 2708),
                    dtype=np.float32)
```

- (np.ones(edges.shape[0])：代表稀疏矩阵要填入的值，为1。若邻接矩阵的相应位置被填入1，则说明两个论文中有引用关系。
- (edges[:, 0], edges[:, 1])：指明了要填入数据的位置，其中edges[:, 0]指明行，edges[:, 1]指明列。
- shape=(labels.shape[0], labels.shape[0])：指明了adj的shape，为 N×NN×N 的矩阵，其中 NN 为样本数。
- dtype=np.float32：指明了矩阵元素的类型。

最终adj的类型为：scipy.sparse.coo.coo_matrix，使用np.array(adj.todense)将其转为ndarray类型的稠密矩阵后，如下：

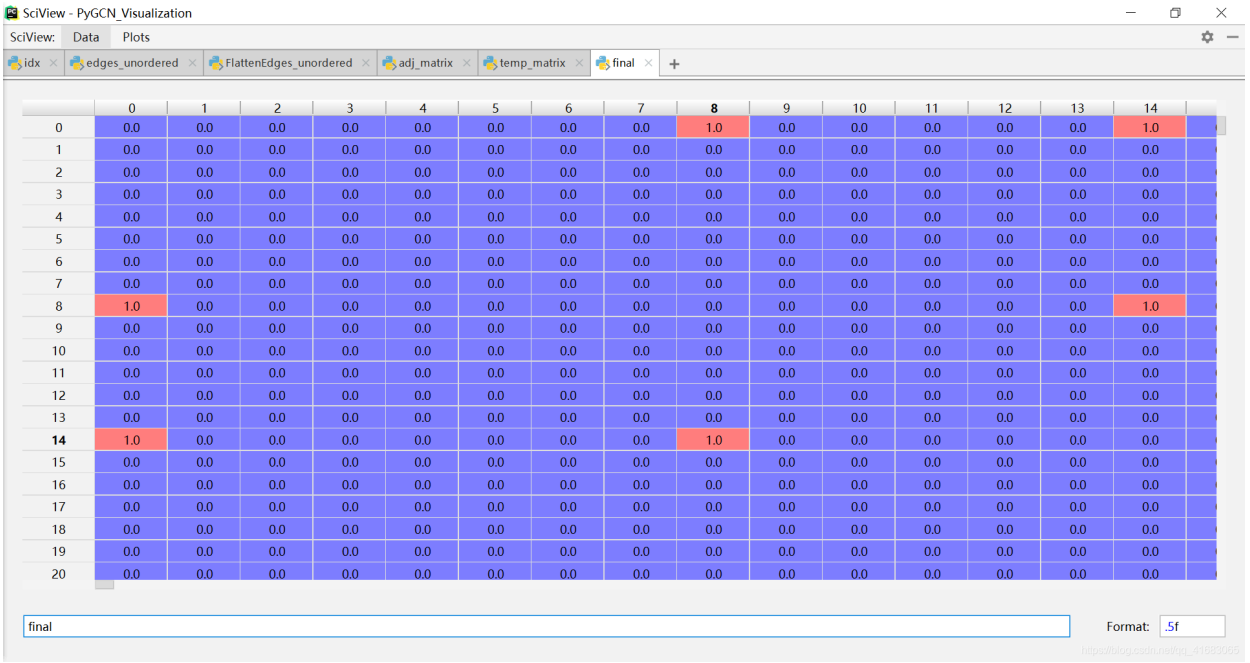


可以看到，由于引用关系的是单向的，导致邻接矩阵adj并不是对称矩阵，构成的Graph也是有向图。

```
# build symmetric adjacency matrix
# np.multiply()函数，数组和矩阵对应位置相乘，输出与相乘数组/矩阵的大小一致
adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

这一步是将非对称的邻接矩阵adj，变为对称矩阵。表现在图上，结果就是将有向图变为无向图。

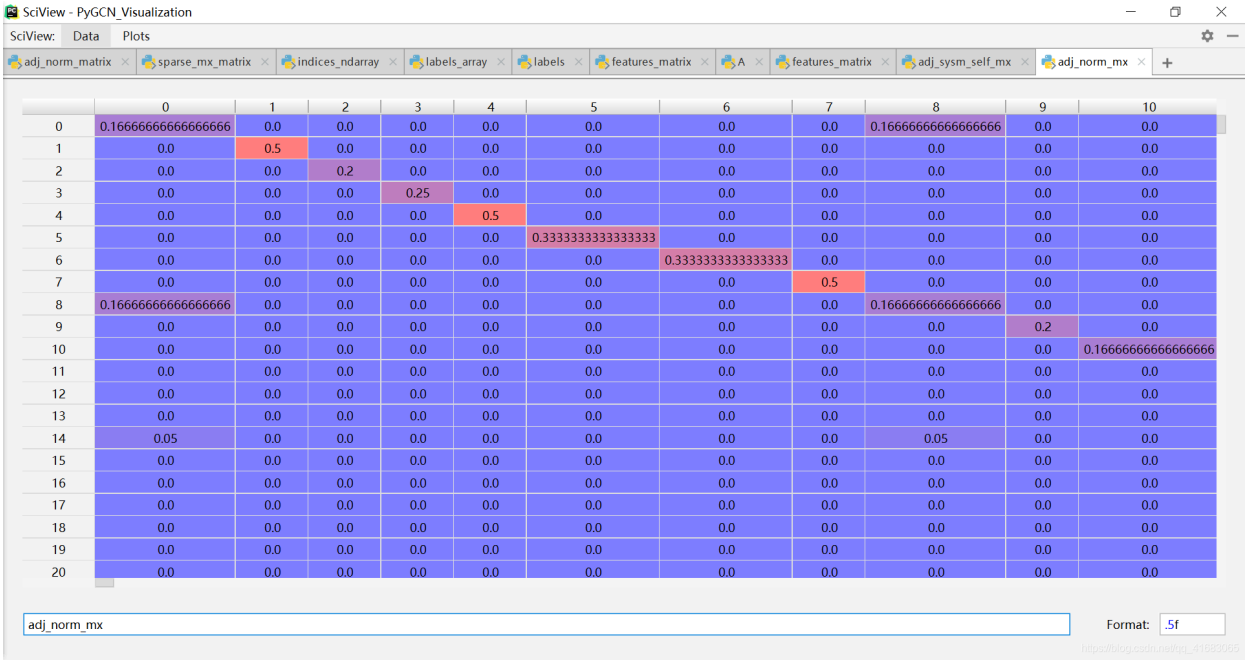
得到的邻接矩阵 adj 如下（可与上面的对比）（如果要使用有向图的邻接矩阵，把这一句注释掉就行）：




```
adj = normalize(adj + sp.eye(adj.shape[0]))    # adj在归一化之前，先引入自环
```

加入自环后的邻接矩阵adj：

归一化后的邻接矩阵adj：



数据集划分

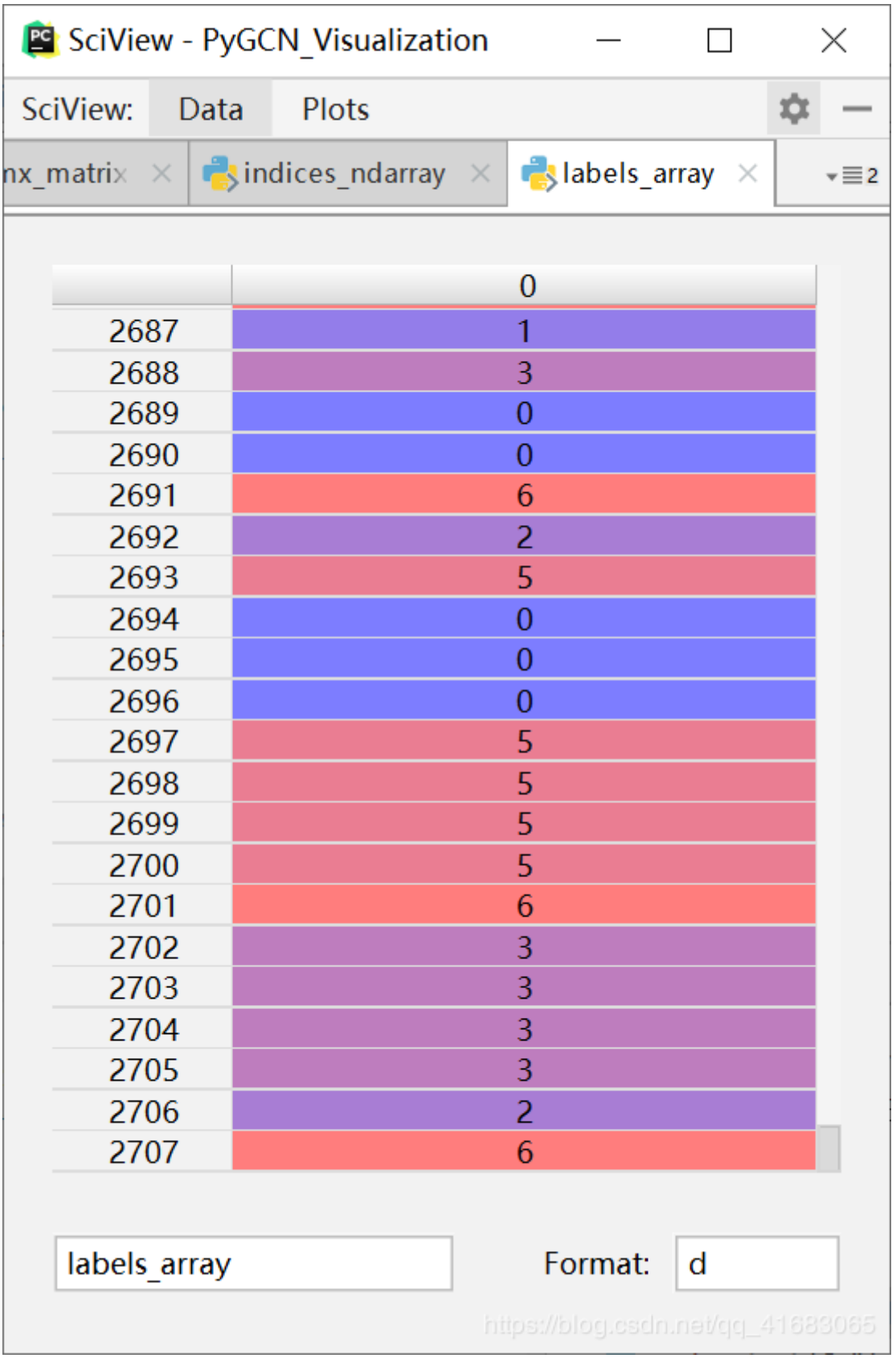
```
# train set, validation set, test set的分组。
idx_train = range(140)
idx_val = range(200, 500)
idx_test = range(500, 1500)
```

按照序号划分数据集。这种划分方式并不是论文中的划分方法。论文中是每一类取相同个数 n 个样本作为训练集。

数据类型转为tensor

```
# 数据类型转tensor
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)
```

- adj：对于邻接矩阵adj的操作，sparse_mx_to_torch_sparse_tensor(adj)，是 Convert a scipy sparse matrix to a torch sparse tensor。具体的细节请看：稀疏矩阵转稀疏张量函数：sparse_mx_to_torch_sparse_tensor(sparse_mx)
- labels：有一点很有意思，是labels的返回值，这个返回值是长这样的：



这就很奇怪。也就是说返回的标签`labels`还是`int`类型的，而不是onehot编码后的结果。这样看来onehot编码并没有起到作用，因为我直接将标签映射到int就可以，而不必须要经过onehot编码这个中间步骤。

在主函数验证，load_data()函数的返回值labels信息如下：

tensor([4, 2, 0, ..., 1, 6, 4])

<class 'torch.Tensor'>
torch.Size([2708])

跟上图显示的一致，即load_data()的返回值labels并不是onehot编码的结果，而是0，1，……，6这样的标签。

- features：是转成了正常的tensor类型（不是adj那样的类型）：

```
tensor([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```


返回数据

```
# 返回数据
return adj, features, labels, idx_train, idx_val, idx_test
```

整个debug load_data()的Demo放到下面了，想尝试的可以拿去用：

```
import numpy as np
import scipy.sparse as sp
from pygcn.utils import
normalize,sparse_mx_to_torch_sparse_tensor,encode_onehot
import torch

'''测试论文编号处理'''
# 读取原始数据集
path="C:/Users/73416/PycharmProjects/PyGCN_Visualization/data/cora/"
dataset = "cora"
idx_features_labels = np.genfromtxt("{}{}.content".format(path, dataset),
                                    dtype=np.dtype(str))

# build graph
# idx_features_labelsidx_features_labels[:, 0]表示取论文编号
idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
```

```
# 通过建立论文序号的序列，得到论文序号的字典
idx_map = {j: i for i, j in enumerate(idx)}

# 读取图的边（论文间的引用关系）
# cora.cites共5429行， 每一行有两个论文编号，表示第一个编号的论文先写，第二个编号的论文引用第一个编号的论文。
edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset),
                                dtype=np.int32)

# 进行一次论文序号的映射
# 论文编号没有用，需要重新的其进行编号（从0开始），然后对原编号进行替换。
# 所以目的是把离散的原始的编号，变成0 - 2707的连续编号
edges = np.array(list(map(idx_map.get, edges_unordered.flatten())),
                 dtype=np.int32).reshape(edges_unordered.shape)

# coo_matrix()：系数矩阵的压缩。分别定义有那些非零元素，以及各个非零元素对应的row和col，最后定义稀疏矩阵的shape。
adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                    shape=(2708, 2708),
                    dtype=np.float32)

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

# 引入自环
adj_sysm_self= adj_sysm + sp.eye(adj.shape[0])

# 归一化
adj_norm = normalize(adj_sysm_self)

features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
features = normalize(features)

labels = encode_onehot(idx_features_labels[:, -1])

# 数据类型转tensor
features = torch.FloatTensor(np.array(features.todense()))
labels = torch.LongTensor(np.where(labels)[1])
adj_norm = sparse_mx_to_torch_sparse_tensor(adj_norm)


# 测试sparse_mx_to_torch_sparse_tensor(sparse_mx)函数
# sparse_mx = adj_norm.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)
```

```
# 增加于2020.3.12，返回非对称邻接矩阵，构成有向图
adj_directed_self=adj+ sp.eye(adj.shape[0])
adj_directed_self_matrix=np.array(adj_directed_self.todense())

adj_directed_norm=normalize(adj_directed_self)
adj_directed_norm_matrix=np.array(adj_directed_norm.todense())
```

计算准确率函数：accuracy(output, labels)

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

输出和输入：

- output为模型model直接的输出，并不是单个的标签（获取预测类别的操作在[accuracy(output, labels)](#计算准确率函数：accuracy(output, labels))中的preds = output.max(1)[1].type_as(labels)实现）。其信息为：

```
tensor([[-5.3865, -5.8370, -5.6641,  ..., -0.0546, -4.6866, -5.4952],
        [-1.9110, -3.6502, -0.8442,  ..., -3.3036, -1.5383, -2.0366],
        [-0.1619, -3.4708, -3.5892,  ..., -3.9754, -3.4787, -3.3948],
        ...,
        [-1.9098, -0.5042, -3.1999,  ..., -3.0369, -3.6273, -2.5525],
        [-2.6523, -2.9252, -2.6154,  ..., -3.0894, -3.3290, -0.3564],
        [-4.6700, -4.5324, -4.5864,  ..., -0.0916, -4.3737, -3.9876]],
       device='cuda:0', grad_fn=<LogSoftmaxBackward>)
<class 'torch.Tensor'>
torch.Size([2708, 7])
```

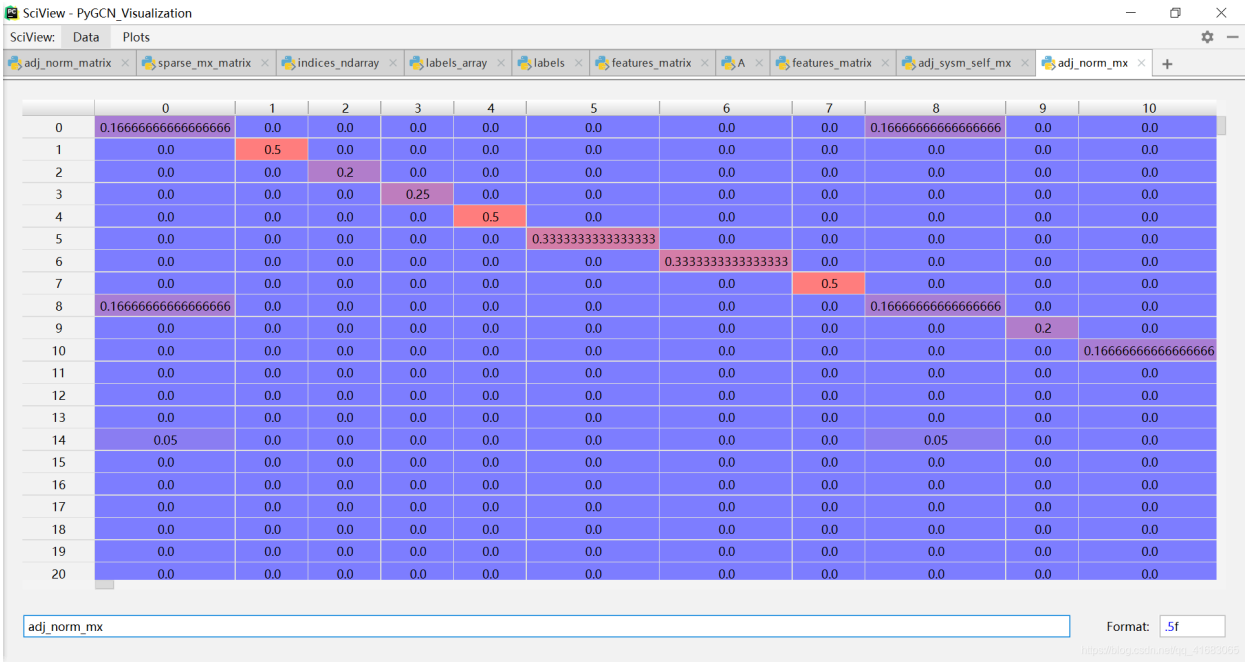
- labels的传入形式：

```
[4 2 0 ... 1 6 4]
<class 'numpy.ndarray'>
(2708,)
```

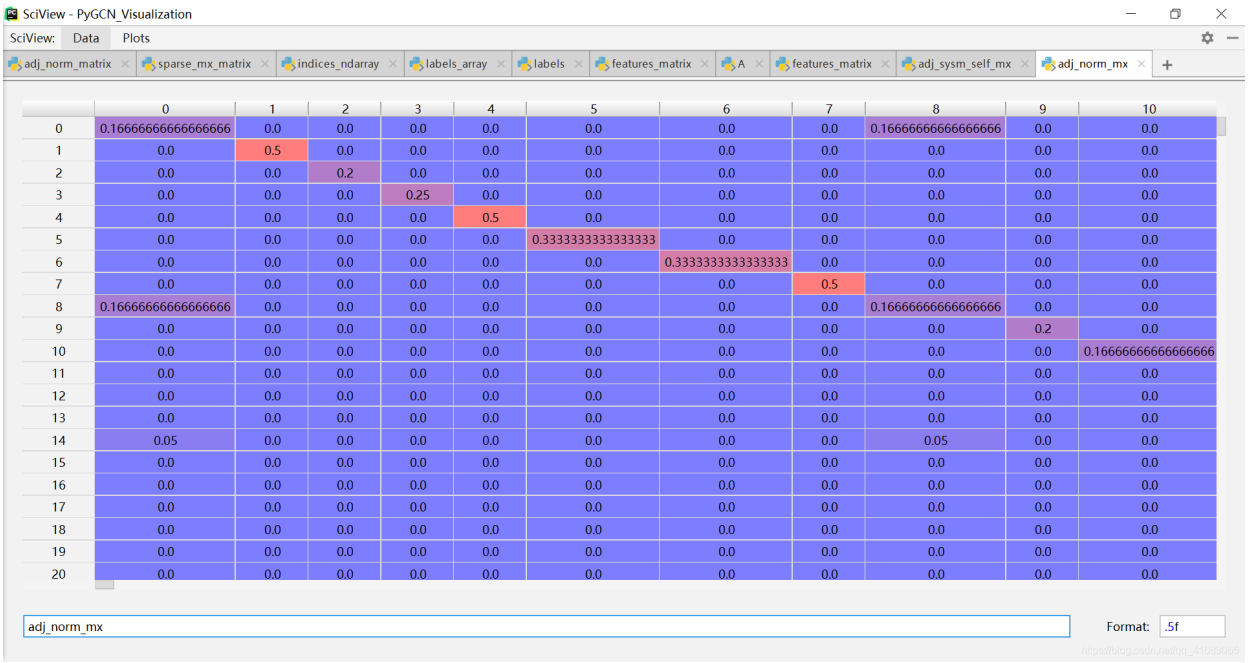
不是onehot编码的格式。

稀疏矩阵转稀疏张量函数：sparse_mx_to_torch_sparse_tensor(sparse_mx)

```
'''稀疏矩阵转稀疏张量'''
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)
```



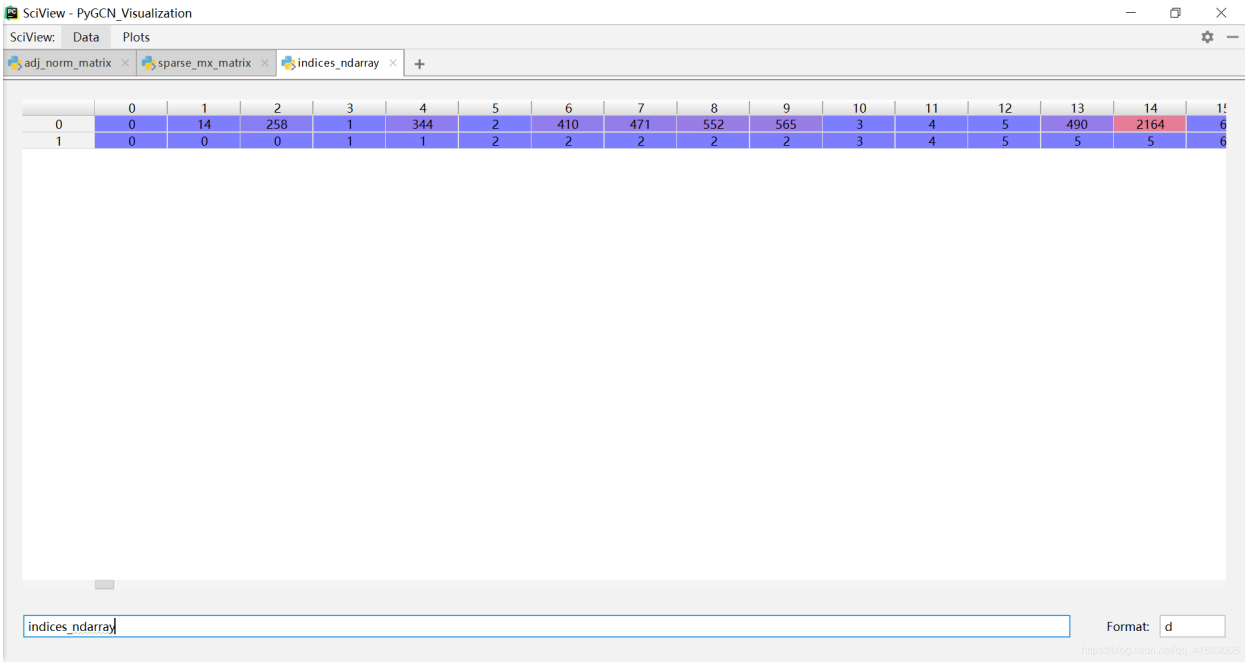
`sparse_mx = sparse_mx.tocoo().astype(np.float32)`之后的`sparse_mx`是长这样的：



也就是说，矩阵还是那个矩阵，只不过通过`.tocoo()`将矩阵的形式变成了COOrdinate format。

csr_matrix.tocoo(*self*, *copy=True*)
Convert this matrix to COOrdinate format.
With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`indices = torch.from_numpy(np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))`这一句是提取稀疏矩阵的非零元素的索引。得到的矩阵是一个[2, 8137]的tensor。
其中第一行是行索引，第二行是列索引。每一列的两个值对应一个非零元素的坐标。



`values = torch.from_numpy(sparse_mx.data)`、`shape = torch.Size(sparse_mx.shape)` 这两行就是规定了数值和shape。没什么好说的。

return torch.sparse.FloatTensor(indices, values, shape)

函数返回值应该注意一下。该函数的返回值的类型是 `torch.Tensor`。
直接打印的结果是一个对象：

print(torch.Tensor)
<class 'torch.Tensor'>

具体查看的话是这样：

```
tensor(indices=tensor([[   0,    14,   258,  ..., 2705, 2706, 2707],
                       [   0,     0,     0,  ..., 2705, 2706, 2707]]),
       values=tensor([0.2500, 0.0500, 0.0833,  ..., 1.0000, 0.2000, 0.2500]),
       size=(2708, 2708), nnz=8137, layout=torch.sparse_coo)
```

说明返回的是一个类的实例！

torch.sparse的官方文档如下：

Torch supports sparse tensors in COO(rdinate) format, which can efficiently store and process tensors for which the majority of elements are zeros.

A sparse tensor is represented as a pair of dense tensors: a tensor of values and a 2D tensor of indices. A sparse tensor can be constructed by providing these two tensors, as well as the size of the sparse tensor (which cannot be inferred from these tensors!)

---

layers.py

全部源代码放到前面：

```
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
    """
    '''定义对象的属性'''
    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))        #
in_features × out_features
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    '''生成权重'''
    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)        # .uniform()：将tensor用从均匀分布
中抽样得到的值填充。
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    '''前向传播 of 一层之内：即本层的计算方法：A_hat * X * W '''
```

```
    def forward(self, input, adj):
        support = torch.mm(input, self.weight)          # torch.mm：Matrix multiply，
input和weight实现矩阵点乘。
        output = torch.spmm(adj, support)               # torch.spmm：稀疏矩阵乘法，sp即
sparse。
        if self.bias is not None:
            return output + self.bias
        else:
            return output

    '''把一个对象用字符串的形式表达出来以便辨认，在终端调用的时候会显示信息'''
    def __repr__(self):
        return self.__class__.__name__ + ' (' \
               + str(self.in_features) + ' -> ' \
               + str(self.out_features) + ')'
```

GraphConvolution是图数据实现卷积操作的层，类似于CNN中的卷积层，只是一个层而已。

GraphConvolution作为一个类，首先要定义其属性：

```
'''定义对象的属性'''
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))          #
in_features × out_features
    if bias:
        self.bias = Parameter(torch.FloatTensor(out_features))
    else:
        self.register_parameter('bias', None)
    self.reset_parameters()
```

主要包括两部分：

- 设定该层中in_features和out_features
- 参数的初始化，通过该对象的reset_parameters()方法实现。参数包括：
    - weight：维度为in_features × out_features
    - bias（if True）：维度为out_features

```
'''生成权重'''
def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
```

```
    self.weight.data.uniform_(-stdv, stdv)          # .uniform()：将tensor用从均匀分布中抽样得到的值填充。
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

就是随机生成权重，不细说了。
但是有一点，生成随机数的种子是可以认为设定的（设定随机数种子），即可以每次初始化得到相同的初始化参数，从而使得结果可复现。

```
'''前向传播 of 一层之内：即本层的计算方法：A * X * W '''
def forward(self, input, adj):
    support = torch.mm(input, self.weight)          # torch.mm：Matrix multiply，input和weight实现矩阵点乘。
    output = torch.spmm(adj, support)               # torch.spmm：稀疏矩阵乘法，sp即sparse。
    if self.bias is not None:
        return output + self.bias
    else:
        return output
```

这一层是定义的本层的前向传播，即本层的计算方法：$A*X*W$ 。
`support = torch.mm(input, self.weight)`是`input`和`weight`实现矩阵乘法，
即 $support=X*W$。
`output = torch.spmm(adj, support)`，由于`adj`是torch.sparse的对象，所以要使用稀疏矩阵乘法`torch.spmm()`，实现的功能是得
到 $outpuy=A*support$。
然后再加上`bias` （ if True ），就得到了本层最后的输出。

```
'''把一个对象用字符串的形式表达出来以便辨认，在终端调用的时候会显示信息'''
def __repr__(self):
    return self.__class__.__name__ + ' (' \
           + str(self.in_features) + ' -> ' \
           + str(self.out_features) + ')'
```

---

models.py

全部源代码放到前面：

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

```
'''GCN类'''
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.gc1 = GraphConvolution(nfeat, nhid)        # 第一层
        self.gc2 = GraphConvolution(nhid, nclass)       # 第二层
        self.dropout = dropout                          # 定义dropout

    '''前向传播 of 层间：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax'''
    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)
```

- `class GCN(nn.Module)`定义了一个图卷积神经网络，在这里有两个卷积层。

该对象的属性为：

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

    self.gc1 = GraphConvolution(nfeat, nhid)        # 第一层
    self.gc2 = GraphConvolution(nhid, nclass)       # 第二层
    self.dropout = dropout                          # 定义dropout
1
```

- gc1：in_feature = nfeat，为数据的原始的feature。out_feature = nhid。
- gc2：in_feature = nhid。out_feature = nclass，为最后待分类的类别数。
- dropout

整个网络的前向传播为：

```
'''前向传播 of 层间：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax'''
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)
```

整个网络的前向传播：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax

```
    super(GCN, self).__init__()

        self.gc1 = GraphConvolution(nfeat, nhid)        # 第一层
        self.gc2 = GraphConvolution(nhid, nclass)       # 第二层
        self.dropout = dropout                          # 定义dropout

    '''前向传播 of 层间：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax'''
    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)
```

`class GCN(nn.Module)`定义了一个图卷积神经网络，在这里有两个卷积层。

该对象的属性为：

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

    self.gc1 = GraphConvolution(nfeat, nhid)        # 第一层
    self.gc2 = GraphConvolution(nhid, nclass)       # 第二层
    self.dropout = dropout                          # 定义dropout
```

- `gc1`：`in_feature` = nfeat，为数据的原始的feature。`out_feature`= nhid。
- `gc2`：`in_feature` = nhid。`out_feature` = nclass，为最后待分类的类别数。
- `dropout`

整个网络的前向传播为：

```
'''前向传播 of 层间：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax'''
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)
```

整个网络的前向传播：整个网络的前向传播的方式：relu(gc1) --> dropout --> gc2 --> log_softmax

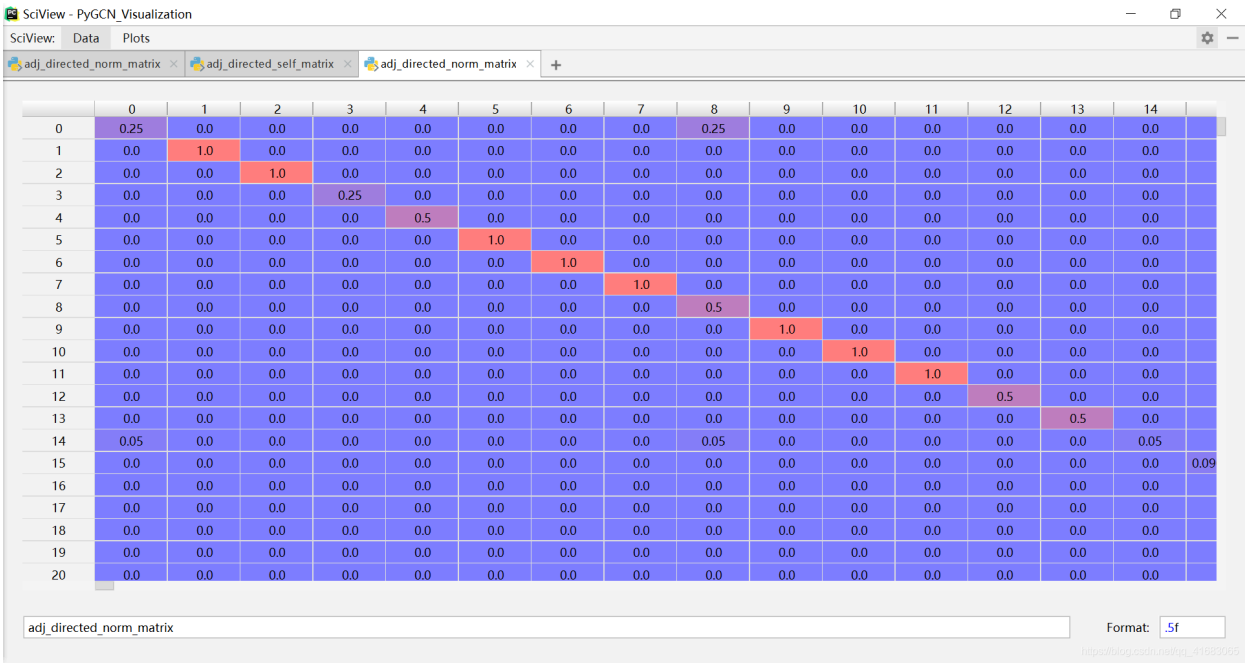
2020.03.12修改，使用有向图邻接矩阵：

注释掉

```
adj=adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

就行了。

得到的归一化后的adj如下：



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
0	0.25	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.25	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.25	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
5	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
6	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	
13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	
14	0.05	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.05	0.0	0.0	0.0	0.0	0.0	0.05	
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.09
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
17	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

可以看到邻接矩阵是非对称的，其所对应的图是有向图。

最后代码的运行结果如下：

```
Test set results: loss= 1.0882 accuracy= 0.6610
```

结果要差于使用对称邻接矩阵的结果，即将图构建为无向图。