前置内容

[0] 频域卷积的主要理论是图傅里叶变换和图的拉普拉斯算子。傅里叶变换可以将函数从空域变到频域，而卷积与傅里叶变换有个等式：

$$(f * g)(t) = F^{-1}[F(f(t)) \odot F[g(t)]]$$

拉普拉斯算子的物理意义是空间二阶导，是标量梯度场中的散度，用了描述物理量的流入流出，这对比图中节点与节点之间的联系。

在图上，拉普拉斯矩阵$L = D - A$，其中D是度矩阵，A是邻接矩阵。

Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

因为无向图中L为对称阵，所以L可以进行矩阵对角化：

$$U^T L U = \Lambda$$

其中U是L特征向量组矩阵，Λ是L的特征值矩阵，对角线上是特征值，其余位置全为0。因此L可以被分解为：

$$L = U \Lambda U^T$$

假设$U = (u_1, \dots, u_n)$，那么图上的频域卷积公式可以写成：

$$\hat{f}(x) = \sum_{n=1}^{N} f(n) u_t(n)$$

那么整张图上的卷积就可以看作：

$$\hat{f} = \begin{bmatrix} \hat{f}(1) \\ \dots \\ \hat{f}(N) \end{bmatrix} = U^T f$$

那么根据卷积和傅里叶变换的等式：

$$(f *_G g) = U(U^T f \odot U^T g) = U(U^T g \odot U^T f)$$

如果将$U^T g$看作时可以学习的卷积核$g(\theta)$，那么图上的卷积公式为：

$$o = f *_G g = U g(\theta) U^T f$$

由此有两个比较经典的频域卷积网络被提出：

- Spectral CNN

  我们上面推导的这个 g_θ 就是首个提出的频域卷积神经网络的卷积核[15]。假设l层的隐藏状态为$h^l \in R^{N \times d_l}$，类似地，第$l+1$层为$h^{l+1} \in R^{N \times d_{l+1}}$。频域卷积层的状态更新计算公式如下：

  $$h_{:,j}^{l+1} = \sigma(U \sum_{i=1}^{d_l} \Theta_{i,j}^l U^T h_{:,i}^l)$$

  $$\Theta_{i,j}^l = g_\theta = \begin{bmatrix} \theta_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \theta_N \end{bmatrix}$$

- ChebNet

基本的频域卷积网络要计算拉普拉斯矩阵所有的特征值和特征向量，计算量巨大。在论文[16]中提出了切比雪夫网络，它应用**切比雪夫多项式(Chebyshev polynomials)**来加速特征矩阵的求解。假设切比雪夫多项式的第k项是T_k,频域卷积核的计算方式如下：

> 切比雪夫多项式是以递归方式定义的一系列正交多项式序列。
>
> $$g_\theta = \sum_{k=0}^{K-1}\theta_k T_k(\tilde{\Lambda}), \text{where } \tilde{\Lambda} = \frac{2\Lambda}{\lambda_{max}} - I_N$$

那么T_k怎么来呢，可以由切比雪夫多项式的定义得来:$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$，递推式的前两项为$T_0(x) = 1$以及$T_1(x) = x$。$\tilde{\Lambda}$的作用是让特征向量矩阵归一化到$[-1, 1]$之间。

Semi-Supervised Classification with GCN [1] [5] [6]

背景

在本文中，作者通过ChebNet的一阶近似推导出逐层更新的图卷积网络并将其称之为GCN，并把GCN应用于半监督分类任务中。

之前关于拉普拉斯矩阵分解的公式还可以表示为:

$$L = U\Lambda U^T = I_N - D^{-1/2}AD^{-1/2}$$

谱卷积由此定义为，信号$x \in \mathbb{R}^N$乘以傅立叶域的滤波器$g_\theta = diag(\theta)$:

$$g_\theta * x = Ug_\theta U^{\mathrm{T}}x$$

这个等式由于需要算L的特征向量，并且计算特征向量的乘法是$O(N^2)$的，因此研究者提出通过切比雪夫多项式$T_k(x)$近似$g_\theta(\Lambda)$，K阶多项式就能取得很好的效果：

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K}\theta'_k T_k(\tilde{\Lambda})$$

因此卷积公式可以表示为：

$$g_{\theta'} \star x \approx \sum_{k=0}^{K}\theta'_k T_k(\tilde{L})x$$

逐层线性模型

限制$K = 1$，即谱卷积近似为一个关于L的线性函数。然后通过堆叠多层来得到卷积能力。继续限制$\lambda_{max} \approx 2$,即特征值被限制在$[0, 2]$中。

$$g_{\theta'} \star x \approx \theta'_0 x + \theta'_1 (L - I_N) x = \theta'_0 x - \theta'_1 D^{-\frac{1}{2}}AD^{-\frac{1}{2}}x$$

上面两个参数θ'，将其看作共享参数并令$\theta'_0 = -\theta'_1$，以便减少计算量：

$$g_\theta \star x \approx \theta\left(I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right)x$$

为了防止重复执行这样的操作，特征值带来的梯度弥散/爆炸等情况，引入正则化：

$$I_N + D^{-1/2}AD^{-1/2} \to \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$$

其中：

- $\hat{A} = A + I_N$
- $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$

将上述公式进一步泛化，$X \in \mathbb{R}^{N \times C}$，即$N$个节点，每个节点特征$C$维。卷积包含$F$个特征映射：

$$Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$$

$\theta \in \mathbb{R}^{C \times F}$是参数矩阵。

由于神经网络是多层，将X记作隐藏特征H，θ看作参数W，因此最终GCN的表示为：

$$H^{(l+1)} = \sigma(\hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)})$$

即**每个节点拿到邻居节点信息然后聚合到自身embedding上**

数据格式

- 邻接矩阵 Adj： $N \times N$， coo_matrix $\rightarrow$ torch.sparse.Tensor
- 特征矩阵 X： $N \times d$， csr_matrix $\rightarrow$ torch.Tensor
- 标签 Labels：str $\rightarrow$ $N \times nclasses$ $\rightarrow$ $N \times 1$ Tensor
- 训练集索引 idx_train
- 验证集索引 idx_val
- 测试集索引 idx_test

代码分析

Model

```python
import torch
from torch import nn
from layer import GCNconv
from torch.functional import F
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.conv1 = GCNconv(nfeat, nhid)
        self.conv2 = GCNconv(nhid, nclass)
        self.dropout = dropout

    def forward(self,features, adj):
        #经过两个卷基层，中间用relu做激活函数，使用dropout
        x = self.conv1(features, adj)
        x = F.relu(x)
        x = F.dropout(x, self.dropout, self.training)
        x = self.conv2(x, adj)
        # 返回softmax
        return F.log_softmax(x, dim=1)
```

$$softmax(X) = \frac{e^X}{\sum_i e^{X_i}}$$
$$X \rightarrow Array$$

Layer

```python
import math
import torch
```

```
from torch import nn
class GCNconv(nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        """
        初始化参数
        输入维（特征的维数），输出（隐藏）维
        初始化Weight和Bias
        """
        super(GCNconv, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weights = 
nn.Parameter(torch.FloatTensor(in_features,out_features))
        if bias:
            self.bias = nn.Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias',None)
        self.reset_parameters()

    def reset_parameters(self):
        """
        生存参数正负行维数之间的均匀分布来初始化参数
        """
        stdv = 1. / math.sqrt(self.weights.size(1))
        self.weights.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input, adj):
        # XW 线性变换
        support = torch.mm(input, self.weights)
        # AXW 左乘邻接矩阵，更新隐层
        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) + ')'
```

Load_data

```
def load_data(path='./data/cora/',dataset='cora'):
    """Load citation network dataset"""
    # 通过np.genfromtxt读进来，格式是一个数组，第一列是idx，最后一列是
label，中间是特征矩阵
    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])

    # bulid graph
    idx = np.array(idx_features_labels[:,0], dtype=np.int32)
    idx_map = {j:i for i, j in enumerate(idx)}
    # 将边读进来 node -- node
    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)
    # 将邻接矩阵变成稀疏矩阵形式，输入：填充值，row，col
    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)

    #Row-normlize
    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]) #equivalent
arr.nonzero()
    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 encode_onehot(labels):
    """onehot 编码"""
    classes = set(labels)
    onehot_dict = {c: np.identity(len(classes))[i, :] for i, c in
enumerate(classes)}
    labels_onehot = np.array(list(map(onehot_dict.get,
labels)),dtype=np.int32)
    return labels_onehot

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
```

```python
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0
    r_mat_inv = sp.diags(r_inv)
    #左×归一化矩阵.
    mx = r_mat_inv.dot(mx)
    return mx

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))
#edge_index
    values = torch.from_numpy(sparse_mx.data) #data
    shape = torch.Size(sparse_mx.shape) #size
    return torch.sparse.FloatTensor(indices,values,shape)
```

Train and Test

```python
def train(epoch):
    t = time.time()
    # 声明是train模式
    model.train()
    optimzer.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()
    optimzer.step()

    # 声明是验证模式, 在此模式下可以验证和测试
    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()))
```

$$nll\_loss = mean(-output(labels))$$

接口学习

1. [4] np.genfromtxt(*fname*, *dtype=*, *comments='#'*, *delimiter=None*, *skip_header=0*, *skip_footer=0*, *converters=None*, *missing_values=None*, *filling_values=None*, *usecols=None*, *names=None*, *excludelist=None*, *deletechars=" !#$%&'()*+, -./:;<=>?@[]^{|}~"*, *replace_space='_'*, *autostrip=False*, *case_sensitive=True*, *defaultfmt='f%i'*, *unpack=None*, *usemask=False*, *loose=True*, *invalid_raise=True*, *max_rows=None*, *encoding='bytes'*)

   从text读取文件，类型为ndarray

2. [3] scipy.sparse(高效的进行矩阵运算)

   - 行压缩矩阵 scipy.sparse.csr_matrix(arg1, shape=None, dtype=None, copy=False)

     构造形式：

     - csc_matrix(D)　 D dim<=2
     - csc_matrix((data, indices, indptr), [shape=(M, N)])

       indptr数组中最后一个元素等于data数组的长度 indptr数组长度减1等于矩阵的行数

       对于矩阵第i行其列索引编号：indices[indptr[i]:indptr[i+1]]；对于矩阵第i行其索引列对应的数据：data[indptr[i]:indptr[i+1]]

     - csr_matrix((data, (row_ind, col_ind)), [shape=(M, N)])

       data表示矩阵填充值，row和col表示边的两个顶点

     - csr_matrix((M, N), [dtype])

       创建空矩阵

     - csr_matrix(S)

   - 列压缩矩阵 scipy.sparse.csc_matrix，构造方式同上
   - scipy.sparse.coo_matrix(arg1，shape = None，dtype = None，copy = False)

     - coo_matrix(D)
     - coo_matrix(S)
     - coo_matrix(data, (row, col))

     主要优点:促进稀疏格式之间的快速转换,to_csr()、to_csc()、to_dense()转换成scr、scc或者稠密矩阵。

   - dok_matrix 继承自dict，key是(row,col)二元组，value为非0元素。构造方式同coo_matrix

     主要优点：非常高效地添加、删除、查找元素，并可转换为其它稀疏矩阵。

   - lil_matrix构造方式同上

     主要用来快速构建稀疏矩阵，但运算切片慢，可转为其它稀疏矩阵进行运算。

3.
```
# 将有向图adj转换为无向图adj即对称adj的方法
adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

4. np.where(arr)相当于arr.nozero()返回非0的行列标

5. [2] torch-sparse: https://ptorch.com/docs/1/torch-sparse

6. [2] model.train() and model.eval()
   - model.train():将模型设置成`training`模式

     仅仅当模型中有`Dropout`和`BatchNorm`是才会有影响。
   - model.eval():将模型设置成`evaluation`模式

     仅仅当模型中有`Dropout`和`BatchNorm`是才会有影响。

参考

[0] https://www.cnblogs.com/SivilTaram/p/graph_neural_network_2.html

[1]https://archwalker.github.io/blog/2019/06/01/GNN-Triplets-GCN.html

[2]https://ptorch.com/docs/8/

[3]https://docs.scipy.org/doc/scipy/reference/sparse.html

[4]https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html

[5]https://github.com/tkipf/pygcn

[6]https://arxiv.org/abs/1609.02907