Graph Attention Network的实现参考

本部分对GAT上的实现方法进行探讨。

1 数据和数据读入

1.1 总体处理结构

数据的读入方法定义在utils.py当中。函数如下：

```
import numpy as np
import scipy.sparse as sp

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))
    # 对文件进行总体的读入

    features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])
    # 获取所有数据的one-hot标签

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    # 获取id的顺序列表

    edges_unordered = np.genfromtxt("{}{}.cites".format(path, dataset), dtype=np.int32)
    # 获取边数据
    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)
    # 获取领接矩阵

    features = normalize_features(features)
    adj = normalize_adj(adj + sp.eye(adj.shape[0]))
    # 获取标准化的初始特征

    idx_train = range(140)
    idx_val = range(200, 500)
```

```
    idx_test = range(500, 1500)

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

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)
    idx_test = torch.LongTensor(idx_test)
    # 转换成torch的形式

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

1.2 one-hot编码和数据初始化

1.2.1 one-hot编码

代码如下：

```
def encode_onehot(labels):
    # The classes must be sorted before encoding to enable static class encoding.
    # In other words, make sure the first class always maps to index 0.
    classes = sorted(list(set(labels)))
    # 首先，获取所有的标签的集合
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
enumerate(classes)}
    # 做一个这样的字典：
    # {'A': array([1., 0., 0.]), 'B': array([0., 1., 0.]), 'C': array([0., 0.,
1.])}
    # 即，将每一个标签直接和对应的one-hot向量映射起来

    labels_onehot = np.array(list(map(classes_dict.get, labels)), dtype=np.int32)
    # 将标签和对应的向量绑定起来

    return labels_onehot
```

在上述的代码当中，有两个值得关注的API。其一是：

```
np.identity(int)
```

它将返回一个单位矩阵。一个例子如下：

```
print(np.identity(3))
>>>
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

上述代码的第6行实际上是一种比较好的写法：它直接性的将标签和one-hot编码对应了起来。

第二个是python的map关键字。它有两个参数：第一个是function，用于指示映射进行的操作，第二个是key，用于指示映射的源。上述代码11行的map(classes_dict.get,lablels)意义是，对于labels的每一个元素进行映射操作，而具体的映射函数就是classes_dict.get方法。它将会返回一个list，然后转换成一个array。

1.2.2 初始化

有两个函数在做这件事，如下：

```
def normalize_features(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 normalize_features(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
```

这两个函数的细节不过多的进行解释了，它们其实都是在对矩阵的行进行初始化，比如说，将矩阵的某一行变成这样：

$$[0, 1, 2] \rightarrow [0, 0.333, 0.666]$$

1.3 读入细节

1.3.1 特征读入

首先看上述函数的前三行：

```
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])
```

第一行实际是一个从txt文本当中读入数据的操作，读入的结果将会变成一个ndarray，在此处的shape是(2708,1435)，每一行的格式如下：

```
['31336' '0' '0' ... '0' '0' 'Neural_Networks']
```

其第一行指明了文章的id号，中间的东西是词袋特征，最后一行是它的类型。

第二行是将词袋特征变换成为一个csr_matrix，这里的csr_matrix表示的是一个稀疏矩阵。它在逻辑上应当和普通的矩阵是一样的。

第三行就是创建了one-hot表示。

1.3.2 创建图

```
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)
```

第一二行是把图当中所有的id弄出来，然后按照索引建立字典。建立出来的idx_map大概长这样：

```
{3221:1,7322:2,...}
```

第四行就是简单的将边数据读入进来。第五行是创建边数据的关键。其中，flatten的操作是将一个矩阵（或者张量）转化成一维数组的形式，比如说，将下面这个矩阵：

```
[[3221 7332],
[322  125]]
```

转化成

```
[3221 7332 322 125]
```

的格式，然后利用map当中的函数进行映射，再reshape回去。其实这里的操作，说白了就是将图当中边的一种表达形式映射成为另一种表达形式。（这里的图上的边有两种表达形式，第一种是以原数据当中的标号来的，第二种是给他们重新按照0，1，2，3这样进行编号。）

在第六行当中使用了另外一个稀疏矩阵，它的基本使用方法如下：

```
row = np.array([0,3,1,0])
col = np.array([0,3,1,2])
data = np.array([4,5,7,9])
coo_matrix((data,(row,col)),shape = (4,4)).toarray()
>>>
array([[4,0,9,0],
       [0,7,0,0],
       [0,0,0,0],
       [0,0,0,5]])
```

其实上述就是稀疏矩阵表示的一种方式。其传入的第一个参数是一个元组，其data表示放入其中的数据，其（row,col）表示其分别对应的坐标，比如说（0，0）坐标的数值是4。这样说来它的含义就非常明显了：

邻接矩阵（在有向图情况下）当中为1的元素肯定有edges.shape[0]个，因为有这么多边，而它们的坐标分别是edges[:,0],edges[:,1]

接下来的操作是将领接矩阵变成对称矩阵的版本，如下：

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

上述的矩阵过程比较复杂，但是验证和下面这个比较好理解的东西是一样的：

```
adj + (adj.T > adj)
```

其中，adj.T - adj返回的是一个矩阵。

1.3.3 数据初始化

接下来是数据的初始化过程：

```
features = normalize_features(features)
adj = normalize_adj(adj + sp.eye(adj.shape[0]))
```

它的作用上述已经提到过，就是将每一行的数值按照比例缩小，并且使得求和的结果为1.

1.3.4 转化

```
idx_train = range(140)
idx_val = range(200, 500)
idx_test = range(500, 1500)

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

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
```

上述的前三行划分了训练集、测试集和验证集。第5-7行进行了到tensor的转换。说一下第七行当中的np.where(labels)[1]弄出来的东西是一个一维的向量，记录了每一个样本对应的标签。

1.4 写法和API总结

我觉得上述数据读入部分当中，有几个值得学习的东西。

第一个是genfromtxt函数，可以快速的帮助从txt中读取出矩阵

第二个是两种edges表达方式的构造，如何将第一种edges转换成第二种edges

第三个是邻接矩阵的构造方式：如何快速的构建邻接矩阵

2 模型

2.1 GraphAttentionLayer

2.1.1 层的初始化

初始化函数如下：

```
    def __init__(self, in_features, out_features, dropout, alpha, concat=True):
        super(GraphAttentionLayer, self).__init__()
        self.dropout = dropout
        self.in_features = in_features
        self.out_features = out_features
        # 输入特征和输出特征的维度数量

        self.alpha = alpha
        # alpha用于构建ReLUctant函数
        self.concat = concat
        # 该参数用于指示是否需要进行向量的拼接

        self.W = nn.Parameter(torch.empty(size=(in_features, out_features)))
        nn.init.xavier_uniform_(self.W.data, gain=1.414)
        # 这里的w是文献当中的转移矩阵，用于将向量从原空间转换到目标空间当中
```

```
16
17          self.a = nn.Parameter(torch.empty(size=(2*out_features, 1)))
18          # 这里的a其实就是文献当中的注意力向量
19
20          nn.init.xavier_uniform_(self.a.data, gain=1.414)
21
22          self.leakyrelu = nn.LeakyReLU(self.alpha)
```

接下来看forward函数。在看它之前，可以先回顾一下做注意力使用的关键步骤：

$$\alpha_{ij} = \frac{exp(LeakyReLU(a[Wh_i||Wh_j]))}{\sum_{k\in\mathcal{N}_i} expLeakyReLU(a[Wh_i||Wh_k])}$$

而：

$$h_i' = \sigma(\sum_{j\in\mathcal{N}_i} \alpha_{ij} Wh_j)$$

想办法合并一下向量，如下：

$$H' = \begin{pmatrix} h_1' \\ h_2' \\ \vdots \\ h_N' \end{pmatrix} = \sigma \begin{pmatrix} \sum_{j\in\mathcal{N}_1} \alpha_{1j} Wh_j \\ \sum_{j\in\mathcal{N}_2} \alpha_j Wh_j \\ \vdots \\ \sum_{j\in\mathcal{N}_N} \alpha_{Nj} Wh_j \end{pmatrix} = \sigma \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1N} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{N1} & \alpha_{N2} & \cdots & \alpha_{NN} \end{pmatrix} \begin{pmatrix} Wh_1 \\ Wh_2 \\ \vdots \\ Wh_N \end{pmatrix}$$

而，对于矩阵$(\alpha_{ij})N \times N$当中的元素，如果它不是目标节点的邻居节点，那么其数值为0。否则，应当进行softmax操作。即：

$$\alpha_{ij} = \begin{cases} 0, if\ adj(i,j) = 0, \\ softmax(a[Wh_i||Wh_j]), else \end{cases}$$

而第二个分支可以表示为：

$$a[Wh_i\ part] \cdot Wh_i + a[Wh_j\ part] \cdot Wh_j$$

即，假设将上述的两个部分记作e_i, e_j，则整个矩阵可以写作：

$$\begin{pmatrix} e_1+e_1 & e_1+e_2 & \cdots & e_1+e_N \\ e_2+e_1 & e_2+e_2 & \cdots & e_2+e_N \\ \cdots & \cdots & \ddots & \cdots \\ e_N+e_1 & e_N+e_2 & \cdots & e_N+e_N \end{pmatrix}$$

上述的过程就比较明了了。然后来看代码，如下：

```
1       def forward(self, h, adj):
2           Wh = torch.mm(h, self.W) # h.shape: (N, in_features), Wh.shape: (N,
    out_features)
3           # 上述是矩阵的乘法操作，先对h进行变换处理，弄出来是一个(N,out_features)的矩阵
4           e = self._prepare_attentional_mechanism_input(Wh)
5           # 上述，把上面的e给计算出来了
```

```
        zero_vec = -9e15*torch.ones_like(e)
        attention = torch.where(adj > 0, e, zero_vec)
        # torch.where的意思是说，如果对应的adj大于零，则填充e的值。这是计算了初步的attention值
        attention = F.softmax(attention, dim=1)
        # 上述对attention进行softmax操作
        attention = F.dropout(attention, self.dropout, training=self.training)
        h_prime = torch.matmul(attention, Wh)
        # 将上一轮当中的东西按照attention的值给加起来
        # 输出的东西是一个(N*out_features)的矩阵

        if self.concat:
            return F.elu(h_prime)
        else:
            return h_prime
        # 上述是一个激活函数的选择
        # 如果要做contcat，表示这个attention是一个中间层，尚且需要一个激活函数
        # 否则它就是一个输出层，不需要激活函数
        # 整个attention层可以看做一个in_feature到out_feature的变换，最后的输出是一个(N*out_feature)的矩阵，也就是每一个节点的新表达方式

    def _prepare_attentional_mechanism_input(self, Wh):
        # Wh.shape (N, out_feature)
        # self.a.shape (2 * out_feature, 1)
        # Wh1&2.shape (N, 1)
        # e.shape (N, N)
        Wh1 = torch.matmul(Wh, self.a[:self.out_features, :])
        Wh2 = torch.matmul(Wh, self.a[self.out_features:, :])
        # broadcast add
        e = Wh1 + Wh2.T
        # 上述式子对应的是e的计算
        return self.leakyrelu(e)
```

2.2 Model

上述的计算过程已经非常明显了，接下来来看整个模型的写法。

代码如下：

```
class GAT(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout, alpha, nheads):
        """Dense version of GAT."""
        super(GAT, self).__init__()
        self.dropout = dropout
        # 这个模型有两个attention层

        self.attentions = [GraphAttentionLayer(nfeat, nhid, dropout=dropout, alpha=alpha, concat=True) for _ in range(nheads)]
```

```
        # 第一个多头的attention层
        for i, attention in enumerate(self.attentions):
            self.add_module('attention_{}'.format(i), attention)
        # 注册信息

        self.out_att = GraphAttentionLayer(nhid * nheads, nclass, dropout=dropout,
alpha=alpha, concat=False)
        # 第二个attention层

    def forward(self, x, adj):
        x = F.dropout(x, self.dropout, training=self.training)
        x = torch.cat([att(x, adj) for att in self.attentions], dim=1)
        # 计算第一个attention层当中每一个头的中间向量，并且把它们连起来

        x = F.dropout(x, self.dropout, training=self.training)
        x = F.elu(self.out_att(x, adj))
        # 计算最后一个attention层当中的值，作为类的输出

        return F.log_softmax(x, dim=1)
```

2.3 训练

训练的代码如下：

```
def train(epoch):
    t = time.time()
    # 记录时间
    model.train()
    # 在model有dropout层的情况下，需要调用train方法随机选择一些神经元进行激活

    optimizer.zero_grad()
    output = model(features, adj)
    loss_train = F.nll_loss(output[idx_train], labels[idx_train])
    # 仅仅在训练数据上计算准确度

    acc_train = accuracy(output[idx_train], labels[idx_train])
    loss_train.backward()
    # 然后反向传播

    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.data.item()),
          'acc_train: {:.4f}'.format(acc_train.data.item()),
          'loss_val: {:.4f}'.format(loss_val.data.item()),
          'acc_val: {:.4f}'.format(acc_val.data.item()),
          'time: {:.4f}s'.format(time.time() - t))

    return loss_val.data.item()
```

上述过程其实没有什么比较难以解读的东西。

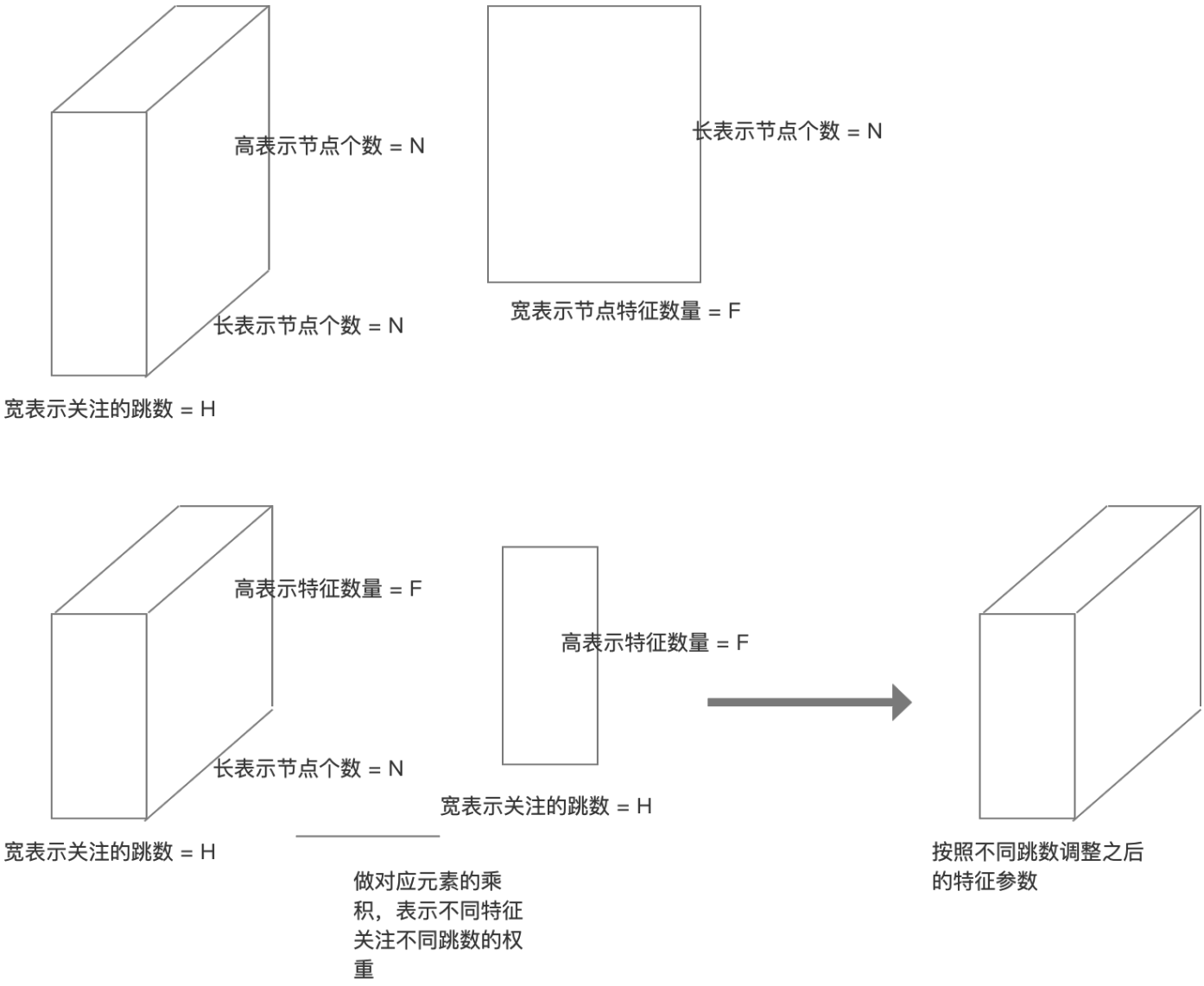
练习：实现DCNN

1 文献回顾

首先，模型的输入是原始节点的向量X，针对于节点分类，其输出是一个$N \times H \times F$的张量Z，Z的具体计算方式如下：

$$Z_{ijk} = f(W_{jk} \cdot \sum_{l=1}^{N_t} P_{ijl} X_{lk})$$

其中i表示节点，j表示距离，k表示feature的索引。上述式子的意思是：经过变换之后的Z的第i个节点的第j跳有k个特征，它第k个特征的数值是距离其有j跳的节点的第k个特征的加权求和，其中权重是转移矩阵的概率。这个数值将会再乘以一个权重，它的权重就是第j跳对第k个特征的权重（这个权重对于每一个节点是一样的），整个过程如下：



然后，直接弄到一个全连接层上去做分类即可。值得注意的是，只要了解其原理，上述的矩阵也没有必要非得如此排列。

2 模型设计

2.1 有问题的版本

由于上述的代码已经给出了进行数据读取的方法，所以仅仅对模型进行修改。如下：

```
class GCNN(nn.Module):
    def __init__(self, input_feature_number,hidden_feature_number,class_number,jump_number = 2):
        super(GCNN, self).__init__()
        self.jump_number = jump_number
        # 首先创建一个转移矩阵，将N*F的原始矩阵按照F*F'转移到N*F'当中
        self.feature_trans = nn.Parameter(torch.empty(size = (input_feature_number,hidden_feature_number)))
        nn.init.xavier_uniform_(self.feature_trans.data, gain=1.414)
```

```
8           self.dense = nn.Parameter(torch.empty(size=
    (class_number,hidden_feature_number*jump_number)))
9           nn.init.xavier_uniform_(self.dense.data, gain=1.414)
10
11
12      def forward(self, x, adj):
13          x = x.matmul(self.feature_trans)
14          Z = torch.stack([(adj**i).matmul(x) for i in range(0,self.jump_number)],0)
15          logits = torch.stack(
16              [self.dense.matmul( Z[:,i,:].reshape(-1)) for i in range(Z.shape[1])],
17              0
18          )
19          return logits
```

上述的代码比较简单，第6和8行分别创建了向量的转移矩阵，以及全连接层。

在forward当中，首先将x转移到目标空间当中，然后堆叠第j步的注意力矩阵，然后用该矩阵计算结果连接全连接层，结果将会变成一个$N \times ClassNumber$的logits矩阵。但是，放进去训练之后，发现loss function竟然是一个负数。这是因为使用了nll_loss function。由于cross_loss = soft_max+log+nll_loss，所以nll_loss是有可能是负数的。于是使用cross_loss替换nll_loss。

为了解决这个问题，不得不先研究一下cross_loss的工作步骤。它是这样的：

首先，输入是一个logits，比如这样：

```
tensor([[ 1.4972, -2.4735, -0.4398],
        [ 0.3252, -0.9270,  0.1815],
        [ 1.3298, -0.8290,  0.4605]])
```

显然，对这个张量当中的每一行，数字越大的项表示对这个项目的认可度越大。然后将这个张量做softmax，变成这样：

```
tensor([[0.8599, 0.0162, 0.1239],
        [0.4647, 0.1328, 0.4025],
        [0.6516, 0.0752, 0.2732]])
```

越大则说明越有信心。之后对每一个东西取对数，变成这样：

```
tensor([[-0.1510, -4.1217, -2.0880],
        [-0.7664, -2.0186, -0.9101],
        [-0.4283, -2.5872, -1.2977]])
```

取了对数之后，由于对数函数也是一个单调递增的函数，也就是说，数字越大，表示越有信心（虽然这个数字是个负数）。然后loss函数将会和真正的标签进行比对，比如说对于下面的这个标签：

```
tensor([0 2 1])
```

然后把第一行的第0个，第二行的第2个，第三行的第1个数字加起来，取反，然后除以3，大概像是这样：

$$\frac{0.1510 + 0.9101 + 2.5872}{3} = 1.2161$$

从上述它的工作原理可以看出，它算出来的loss不可能是负数。但是更换之后，效果还是非常糟糕，完全就是训烂掉。后来发现是模型的编写有问题。在pytorch当中，直接使用*或者是**算出来的结果是矩阵对应元素相乘，而不是矩阵的乘法，所以上述的模型实际上是错误的，应该这样写：

2.2 模型最终版本

```
def forward(self, x, adj):
    # x: N*F ,self.feature_trans:F*F'
    x = x.matmul(self.feature_trans)
    # Z: H*N*F'
    adj_list = []
    for i in range(self.jump_number):
        temp_tensor = adj
        for j in range(i):
            temp_tensor = temp_tensor.matmul(adj)
        adj_list.append(temp_tensor.matmul(x))

    Z = torch.stack(adj_list,0)
    logits = torch.stack(
        [self.dense.matmul( Z[:,i,:].reshape(-1)) for i in range(Z.shape[1])],
        0
    )
    return logits
```

这样就对了。

算出来在cora验证集上的结果大概是最高0.83，测了几组。feature维度不是很影响，但是jump_number = 1的时候大概只有0.8左右。jump_number = 2的时候比较高。