利用 GCN 模型完成节点分类与链路预测

姓名：陈泽豪　学号：SA22001009

2023 年 2 月 15 日

摘要

本报告首先会介绍 GCN 模型的实现内核以及具体的形式，然后利用自己写的 GCNConv 完成 GCN 模型的编写，在得到了 GCN 模型之后对 Cora，Citeseer，PPI 三个数据集完成节点分类与链路预测的任务，并分析自环，层数，DropEdge，PairNorm，激活函数等因素对结果的影响。

一、GCN 网络介绍

1.1 图表示学习

首先介绍什么是图表示学习，简要来说就是给定一个图 G=(V, E)，将图上的节点压缩成低维的向量表示 $R^{n\times k}(k << n)$。在这张图上存在以下的一些性质：

1. 相邻节点具有相似的向量表示。
2. 具有相似属性的节点具有相似向量表示。
3. 节点的顺序变换对向量表示没有影响。

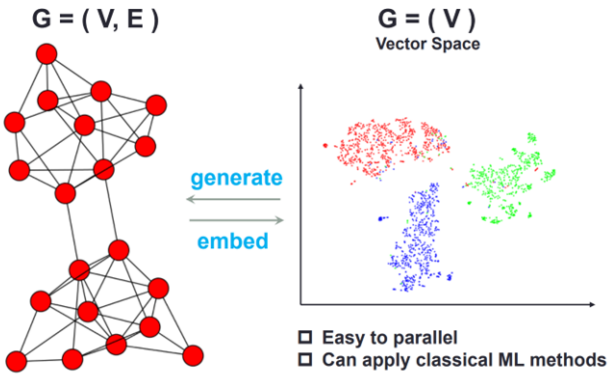


图 1: 高维到低维的映射

我们所要介绍的 GCN 算法就是一种基于深层图神经网络的表示学习方法，它基于频域或谱域 spectral domain 的图卷积网络，可类比到对图片进行傅里叶变换后，再进行乘积。

1.2 拉普拉斯矩阵

以无向图为例，我们定义每个节点处的度矩阵 D 以及整个图的邻接矩阵 A，定义拉普拉斯矩阵 L=D-A。拉普拉斯矩阵具有如下的性质：

1. L 1=D1-A1=d -d=0，说明 1 为特征向量，0 为特征值。
2. L 为对称半正定矩阵。
3. L 的特征值中 0 的个数等同于原图的连通块数量。

由于拉普拉斯矩阵的性质，我们可以对拉普拉斯矩阵做对称归一化，具体如下所示：

$$L^{sym} = D^{-1/2}LD^{-1/2} = I - D^{-1/2}AD^{-1/2} \tag{1}$$

对称归一化之后的矩阵 L^{sym} 具有如下的性质：

1. 对角线上元素为 1，仍为对称矩阵。
2. 0 为矩阵 L^{sym} 的特征值，特征向量为 $D^{1/2}1$。
3. L 的特征值中 0 的个数等同于原图的连通块数量。

1.3 图傅里叶变换

傅里叶变换就是选取了 $\{e^{inx}\}_{n=-\infty}^{+\infty}$ 为基函数，然后将函数从时域在谱域上进行展开，因此图傅里叶变换也是类似的，我们选取合适的基函数并在其上进行展开。令 $L = U\sigma U^T$ 为 L 的正交相似，则可以将 U 矩阵内的所有列向量作为傅里叶变换基傅里叶变换就是将原信号在正交基上展开，因此有：

$$x = U\hat{x}, U = (u_1, u_2, ..., u_n). \tag{2}$$

因此可以有傅里叶变换为 $F(f) = U^T f$，以及傅里叶逆变换为 $F^{-1}(f) = Uf$。

当有了傅里叶变换的知识后，又有图上的卷积相当于图的傅里叶变换乘积可以知道，如果对图上信号做卷积处理，相当于先到谱域进行乘积后回到时域，给定一个图信号 x 和一个卷积核 g，则有：

$$x * g = U(U^T x \odot U^T g). \tag{3}$$

因此如果将 $U^T g$ 视为一个 g_θ 对角阵，便有：

$$x * g = U(U^T x \odot U^T g) = U g_\theta U^T x. \tag{4}$$

1.4 多项式卷积核

首先需知道 L 名字为拉普拉斯矩阵，其作用在信号上相当于一个拉普拉斯算子作用在函数上，它求得是与某一个节点 i 相连的其他节点 j 对 i 的牵引作用。因此我们将 Lf 视为图信号 f 在图上的一次传播。

$$Lx = \sum_{i<j}(x_j - x_i). \tag{5}$$

下面考虑一个已经经过了归一化的 L，认为卷积核与 L 矩阵的相似对角阵相关，即有：

$$g_\theta(\sigma) = \sum_{i=0}^{k} \theta_i \sigma^i. \tag{6}$$

则此时有：

$$x * g = U g_\theta U^T x = \sum_{i=0}^{k} \theta_i U \sigma^i U^T x = \sum_{i=0}^{k} \theta_i L^i x. \tag{7}$$

相当于信号在图上的 k 次传播，一个信号只会取决于以它为中心的 k 层节点。若用切比雪夫多项式的基函数来重新表示卷积阵：

$$g_\theta(\sigma) = \sum_{i=0}^{k} \theta_i T_i(\sigma). \tag{8}$$

由于切比雪夫多项式定义域为 $[-1, 1]$，因此需对此时的 σ 做转化：

$$\hat{g}_\theta = \sum_{i=0}^{k} \theta_i T_i(\hat{\sigma}), \quad \hat{\sigma} = \frac{2\sigma}{\lambda_{max}} - I. \tag{9}$$

同理有：

$$x * g = U \hat{g}_\theta U^T x = \sum_{i=0}^{k} \theta_i T_i(\hat{L}) x, \quad \hat{L} = \frac{L}{2\lambda_{max}} - I. \tag{10}$$

而一个归一化的拉普拉斯矩阵 L^{sym} 最大特征值约为 2，因此有：

$$\hat{L}^{sym} = \frac{2(I - D^{-1/2} A D^{-1/2})}{2} - I = -D^{-1/2} A D^{-1/2}. \tag{11}$$

1.5 从 chebynet 到 GCN

如果上面的切比雪夫多项式只取到 1 阶，且取 $\theta_0 = -\theta_1$，有：

$$x * g = \theta_0 x + \theta_1(-D^{-1/2}AD^{-1/2})x = \theta_0(I + D^{-1/2}AD^{-1/2})x. \tag{12}$$

$$\Rightarrow Y = (I + D^{-1/2}AD^{-1/2})X\Theta \tag{13}$$

其中 Θ 为优化参数，X 为图上信号，其中有 $x \in \mathbf{R}^n$ $X \in \mathbf{R}^{n\times d}$。$\Theta$ 的列数决定了隐藏层内有多少图。虽然只能在图上做一次传播，只与相邻节点有关，但只要叠加多次即可。

由于 $I + D^{-1/2}AD^{-1/2}$ 的特征值在 $[0, 2]$ 之间，多次叠加会爆炸，因此我们采用添加自环的方式来解决，具体来讲就是取 $\tilde{A} = A + I, \tilde{D} = D + I$。最终的 GCN 模型形如：

$$\hat{Y} = f(X, A) = Softmax(\hat{A}ReLU(\hat{A}XW^0)W^1). \tag{14}$$

其中 $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$，$\tilde{A} = A + I_n$，$D_{ii} = \sum_j \tilde{A}_{ij}$。

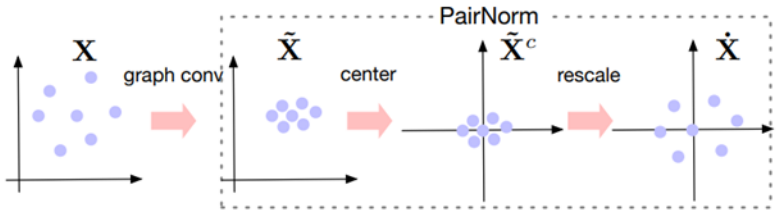
1.6 GCN 的问题：过平滑

使用 GCN 会导致堆叠层数过高时，频率只剩下 0，因此最终的特征向量只剩下 1 向量，这会导致过平滑问题，本身可能只想要一类的节点逐渐趋于一致，但是这样所有节点还没做训练都会趋于一致，无法进行下去。

解决方法一是采用 DropEdge，具体操作如下所示：

1. 随机选择边 $a = Bernoulli(p)$。
2. 稀疏化邻接矩阵 $A_{ij} = A_{ij} * a$。

解决方法二是采用 PairNorm，具体操作如下所示：



主要就是对输出结果做一次中心化后再拉伸。

二、实验配置

本次实验通过 Anaconda 搭建出了虚拟环境，整个代码在 python 3.9.15，cuda 11.3.1，numpy 1.23.4，pytorch 1.12.1 环境下进行实验，GPU 使用的是 NVIDIA Geforce RTX 3060。

三、对数据集的处理

3.1 Cora 数据集

首先介绍以下 Cora 数据集下载之后的各个文件包含着的信息。Cora 数据集包含了机器学习的文章，这些文章可以被分类为以下七个类：Case_Based，Genetic_Algorithms，Neural_Networks，Probabilistic_Methods，Reinforcement_Learning，Rule_Learning，Theory。在最后的语料库中，每篇论文引用或被至少一篇其他论文引用。整个语料库共有 2708 篇论文。在词干和删除词缀后，我们留下了 1433 个独特单词的词汇表。删除所有文档频率低于 10 的单词。即每篇文章的特征维度为 1433 维。在.content 文件内，存储着每篇文章的特征信息以及最后的类别：

$$< paper\_id >< word\_attributes > + < class\_label > \tag{15}$$

在.cites 文件中存储着语料库中的文献引用情况，每一行用 <ID of cited paper> <ID of citing paper> 表示。

下面就是我在 CoraData.py 文件中的类 CoraData 中的读取文件初始化过程：

```
def __dataset_loader(self):
# path = "./cora/cora"
path_cites = self.cora_path + "/cora.cites"
path_contents = self.cora_path + "/cora.content"

with open(path_contents, 'r', encoding='utf-8') as file_content:
for node in file_content.readlines():
node_cont = node.split()
# 按照进入字典的顺序进行重新排列文章
self.index_of_pg[node_cont[0]] = len(self.index_of_pg)
self.feature_of_pg.append([int(i) for i in node_cont[1:-1]])

label = node_cont[-1]
if label not in self.index_of_pg_label.keys():
 # 按照进入字典的顺序进行重新排列label
self.index_of_pg_label[label] = len(self.index_of_pg_label)
self.label_of_pg.append(self.index_of_pg_label[label])

with open(path_cites, 'r', encoding='utf-8') as file_cite:
for edge in file_cite.readlines():
cited, citing = edge.split()
# 本身为有向图，这里设置为无向
edge_fir = [self.index_of_pg[citing], self.index_of_pg[cited]]
edge_sec = [self.index_of_pg[cited], self.index_of_pg[citing]]
```

```
if edge_fir not in self.edge_of_pg:
self.edge_of_pg.append(edge_fir)
if edge_sec not in self.edge_of_pg:
self.edge_of_pg.append(edge_sec)
```

3.2 Citeseer 数据集

需要知道的是 Citeseer 数据集与 Cora 数据集的处理方式几乎一模一样，但是 Citeseer 给出的数据集本身存在问题，一些在.cites 文件中出现的连接边对应的节点在.content 文件中并未出现对此节点 feature，类别的介绍。在 PYG 中采用了在.content 中添加对应节点 feature 的做法，我采用的是将.cites 文件中那些没有节点信息的边删除。因此最终处理方式如下所示：

```
def __init__(self, path_of_citeseer):
self.path_of_citeseer = path_of_citeseer

# 重新将论文从0开始编码
self.index_of_pg = dict()
# 重新将论文标签以数字形式呈现
self.index_of_pg_label = dict()

self.feature_of_pg = []
self.label_of_pg = []
self.edge_of_pg = []

self.__dataset_loader()
self.num_nodes = len(self.feature_of_pg)
self.num_edges = len(self.edge_of_pg)
self.num_of_class = len(self.index_of_pg_label)
self.feature_dim = len(self.feature_of_pg[0])

def __dataset_loader(self):
# path = "./cora/cora"
path_cites = self.path_of_citeseer + "/citeseer.cites"
path_contents = self.path_of_citeseer + "/citeseer.content"

with open(path_contents, 'r', encoding='utf-8') as file_content:
for node in file_content.readlines():
node_cont = node.split()
# 按照进入字典的顺序进行重新排列文章
```

```
self.index_of_pg[node_cont[0]] = len(self.index_of_pg)
self.feature_of_pg.append([int(i) for i in node_cont[1:-1]])

label = node_cont[-1]
if label not in self.index_of_pg_label.keys():
# 按照进入字典的顺序进行重新排列label
self.index_of_pg_label[label] = len(self.index_of_pg_label)
self.label_of_pg.append(self.index_of_pg_label[label])

with open(path_cites, 'r', encoding='utf-8') as file_cite:
for edge in file_cite.readlines():
cited, citing = edge.split()

# 需要注意citeseer是错误的数据集，这里做特殊处理，将.content中不存在的节点对应的边删除
if (cited not in self.index_of_pg.keys()) or (citing not in self.index_of_pg.keys()):
continue

# 本身为有向图，这里设置为无向
edge_fir = [self.index_of_pg[citing], self.index_of_pg[cited]]
edge_sec = [self.index_of_pg[cited], self.index_of_pg[citing]]
if edge_fir not in self.edge_of_pg:
self.edge_of_pg.append([self.index_of_pg[citing], self.index_of_pg[cited]])
if edge_sec not in self.edge_of_pg:
self.edge_of_pg.append([self.index_of_pg[cited], self.index_of_pg[citing]])
```

3.3 PPI 数据集

PPI 数据集相比前两个数据集更加复杂,PPI 网络是蛋白质相互作用（Protein-Protein Interaction,PPI）网络的简称,一般地,如果两个蛋白质共同参与一个生命过程或者协同完成某一功能,都被看作这两个蛋白质之间存在相互作用。多个蛋白质之间的复杂的相互作用关系可以用 PPI 网络来描述。ppi-class_map.json 为节点的 label 文件，shape 为 (121, 56944), 每个节点的 label 为 121 维，因此这个问题是一个多标签的分类问题。ppi_feats.npy 文件保存节点的特征，shape 为 (56944, 50)(节点数目，特征维度)，值为 0 或 1，且 1 的数目稀少。ppi-G.json 文件为节点和链接的描述信息，在此 json 文件中可以得到训练集，验证集，测试集的分类情况以及图上边的连接情况，在处理时首先去除自连接以创建邻接矩阵。具体的处理我在 PPIData.py 中的类 PPIDataFromJson 中进行了处理，代码见下：

```
class PPIDataFromJson:
def __init__(self, path_of_ppi):
self.ppi_path = path_of_ppi
```

```
self.feature_of_pg = None
self.label_of_pg = None

self.edge_of_pg = []
self.train_mask = []
self.test_mask = []
self.val_mask = []
self.num_nodes = 0
self.num_edges = 0
self.num_of_class = 0
self.feature_dim = 0

self.get_node_feature()
self.get_edge_index()
self.get_node_label()

def get_node_feature(self):
path_of_feature = self.ppi_path + "/ppi-feats.npy"
self.feature_of_pg = np.load(path_of_feature)
self.num_nodes = len(self.feature_of_pg)
self.feature_dim = len(self.feature_of_pg[0])
return

def get_edge_index(self):
graph = self.ppi_path + "/ppi-G.json"
with open(graph, 'r', encoding='utf-8') as fp:
json_format = json.load(fp)
for nodes in json_format['nodes']:
test_bool = nodes['test']
node_id = int(nodes['id'])
val_bool = nodes['val']
if test_bool:
self.test_mask.append(node_id)
elif val_bool:
self.val_mask.append(node_id)
else:
self.train_mask.append(node_id)

for edges in json_format['links']:
source = edges['source']
target = edges['target']
```

```
if source != target:
self.edge_of_pg.append([source, target])
self.edge_of_pg.append([target, source])
self.num_edges = len(self.edge_of_pg)
return

def get_node_label(self):
labels = self.ppi_path + "/ppi-class_map.json"
with open(labels, 'r', encoding='utf-8') as fp:
json_format = json.load(fp)
self.num_of_class = len(json_format['0'])
self.label_of_pg = np.ones([len(json_format), len(json_format['0'])], dtype=float)
for label in json_format.keys():
key = int(label)
self.label_of_pg[key] = np.array(json_format[label])
return

def data_partition_node(self):
train_mask_tensor = torch.tensor(self.train_mask, dtype=torch.long)
val_mask_tensor = torch.tensor(self.val_mask, dtype=torch.long)
test_mask_tensor = torch.tensor(self.test_mask, dtype=torch.long)
return train_mask_tensor, val_mask_tensor, test_mask_tensor
```

四、过平滑的处理

这一部分主要就是展示一下过平滑处理的部分代码。

4.1 Dropedge

首先是 Dropedge，Dropedge 告诉我们需要自己设定一个概率，然后按照概率去隐去邻接矩阵中的一些边，当然最终还是需要保持邻接矩阵的对称性，因此最终的代码实现如下，我在 CoraData.py 中的函数 random_adjacent_sampler 做了实现，根据输入是否是对称的边集做了区分：

```
# 得到一个随机隐藏边的邻接矩阵
def random_adjacent_sampler(edge_of_pg, num_graph_node,
drop_edge=0.1, symmetric_of_edge=False):
if not symmetric_of_edge:
```

```
new_edge_of_pg = []
edge_num = int(len(edge_of_pg))
sampler = np.random.rand(edge_num)
for i in range(int(edge_num)):
if sampler[i] >= drop_edge:
new_edge_of_pg.append(edge_of_pg[i])
new_edge_of_pg = np.array(new_edge_of_pg)
new_edge_of_pg = convert_symmetric(new_edge_of_pg)
graph_w = np.ones(len(new_edge_of_pg))
adj = sp.coo_matrix((graph_w, (new_edge_of_pg[:, 0], new_edge_of_pg[:, 1])),
shape=[num_graph_node, num_graph_node])
else:
new_edge_of_pg = []
half_edge_num = int(len(edge_of_pg) / 2)
sampler = np.random.rand(half_edge_num)
for i in range(int(half_edge_num)):
if sampler[i] >= drop_edge:
new_edge_of_pg.append(edge_of_pg[2 * i])
new_edge_of_pg.append(edge_of_pg[2 * i + 1])
new_edge_of_pg = np.array(new_edge_of_pg)
graph_w = np.ones(len(new_edge_of_pg))
adj = sp.coo_matrix((graph_w, (new_edge_of_pg[:, 0], new_edge_of_pg[:, 1])),
shape=[num_graph_node, num_graph_node])
return adj
```

4.2 Pairnorm

对于 Pairnorm，主要有三种实现方式，在 Pairnorm 源码中就有这三种的实现方式，具体如下：

```
def PairNorm(x_feature):
mode = 'PN-SI'
scale = 1
col_mean = x_feature.mean(dim=0)
if mode == 'PN':
x_feature = x_feature - col_mean
row_norm_mean = (1e-6 + x_feature.pow(2).sum(dim=1).mean()).sqrt()
x_feature = scale * x_feature / row_norm_mean

if mode == 'PN-SI':
x_feature = x_feature - col_mean
```

```
row_norm_individual = (1e-6 + x_feature.pow(2).sum(dim=1, keepdim=True)).sqrt()
x_feature = scale * x_feature / row_norm_individual

if mode == 'PN-SCS':
row_norm_individual = (1e-6 + x_feature.pow(2).sum(dim=1, keepdim=True)).sqrt()
x_feature = scale * x_feature / row_norm_individual - col_mean

return x_feature
```

五、节点分类任务实验结果

5.1 Cora 数据集结果

采用自环，不使用 DropEdge，PairNorm，激活函数取得为 relu，设置卷积层层数为 2 层，学习率为 0.01，隐藏层维数取 16 层，设置误差函数为交叉熵函数，设置优化器的 weigut_decay 为 5e-4，epoch 数为 500，误差变化以及在验证集上的精度变化如下所示：

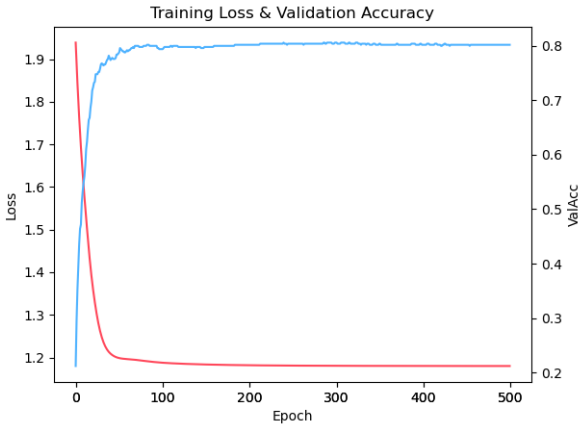


图 2: Cora 数据集节点分类任务

在测试集上的精度结果为 83.1%。

5.2 Citeseer 数据集结果

采用自环，不使用 DropEdge，PairNorm，激活函数取得为 relu，设置卷积层层数为 2 层，学习率为 0.01，隐藏层维数取 16 层，设置误差函数为交叉熵函数，设置优化器的 weigut_decay 为 5e-4，epoch 数为 500，误差变化以及在验证集上的精度变化如下所示：

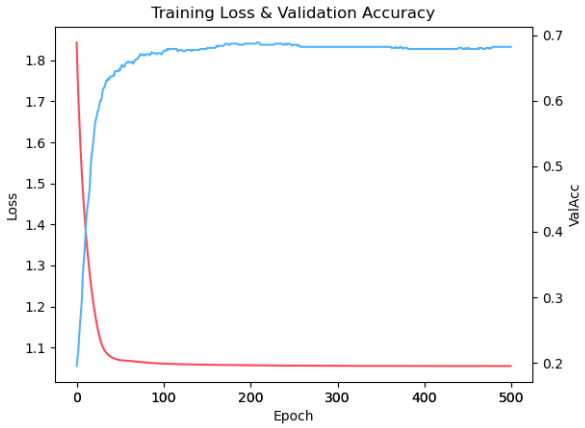


图 3: Citeseer 数据集节点分类任务

最终在测试集上的精度为 69.2%。

5.3 PPI 数据集结果

采用自环，使用 DropEdge 的概率为 0.05，取 PairNorm 的形式为'PN-SI'，激活函数取得为 relu，设置层数为 2 层，学习率为 0.006，隐藏层维数取 512 层，设置误差函数为 BCELoss 函数，不设置优化器的 weigut_decay，epoch 数为 1000，PPI 中使用的精度测量为 micro_f1_score，误差变化以及在验证集上的精度变化如下所示：

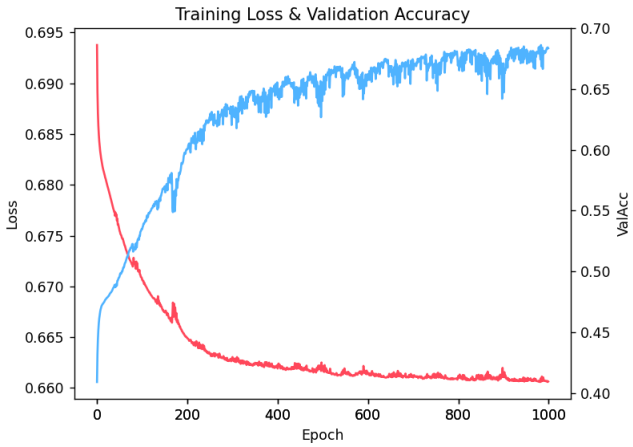


图 4: PPI 数据集节点分类任务

最终在测试集上的精度为 70.94%。

六、链路预测任务实验结果

首先介绍一下链路预测任务，链接预测比节点分类更复杂，因为我们需要使用节点嵌入对边缘进行预测。预测步骤大致如下：

1. 编码器通过处理具有两个卷积层的图来创建节点嵌入。
2. 在原始图上随机添加负链接。这使得模型任务变为对原始边的正链接和新增边的负链接进行二元分类。
3. 解码器使用节点嵌入对所有边 (包括负链接) 进行链接预测 (二元分类)。它从每条边上的一对节点计算节点嵌入的点积。然后聚合整个嵌入维度的值，并在每条边上创建一个表示边存在概率的值。

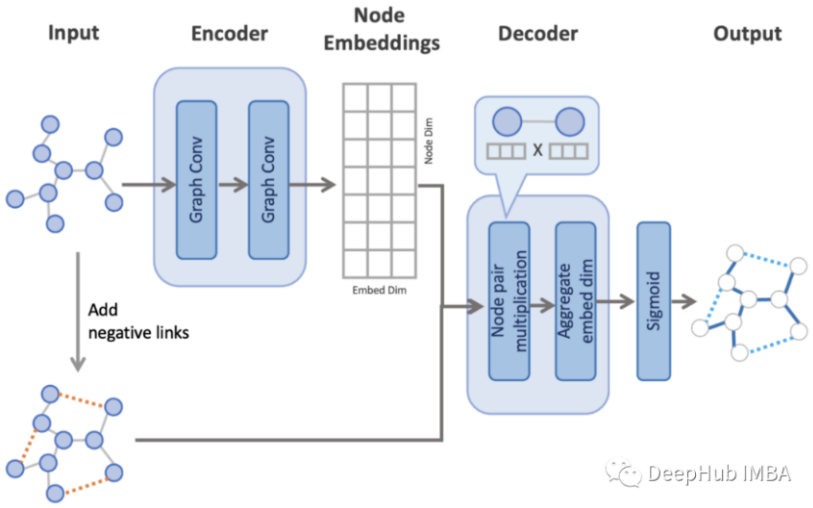


图 5: 链路预测示意图

我实现的链路预测详见 MyGCNNet.py 中的 MyLinkPredictionGCN 类，包含了 encode 部分与 decode 部分。下面给出这部分的代码：

```
# encode部分将num_node*num_feature的in_feature转变成了num_node*out_feature_dim的矩阵
def encode(self, in_feature, adj):
output = in_feature
for i, layer in enumerate(self.layers):
if i != 0:
output = self.dropout(output)
output = layer(output, adj)

if i != (self.num_of_hidden_layer - 1):
output = F.relu(output)
```

```
if self.use_pair_norm:
output = self.PairNorm(output)
return output

# 这里的out_feature是encode部分输出的量，edge_index为一个2E*2的张量
# 下面需要将一个edge上的两个点处的out_feature做点积
# pos_edge_index为E*2，neg_edge_index为E*2，logits为一个2E*1的量，  表示那些边的属性
@staticmethod
def decode(out_feature, pos_edge_index, neg_edge_index):
edge_index = torch.cat([pos_edge_index, neg_edge_index], dim=-1)
logits = (out_feature[edge_index[0]] * out_feature[edge_index[1]]).sum(dim=-1)
logits = torch.sigmoid(logits)
return logits
```

6.1 Cora 数据集结果

采用自环，不使用 DropEdge，PairNorm，激活函数取得为 relu，设置卷积层层数为 2 层，学习率为 0.01，隐藏层维数取 128 层，node_embed 维数为 64，设置误差函数为 binary_cross_entropy_with_logits 函数，设置优化器的 weigut_decay 为 5e-4，epoch 数为 100，误差变化以及在验证集上的精度变化如下所示：

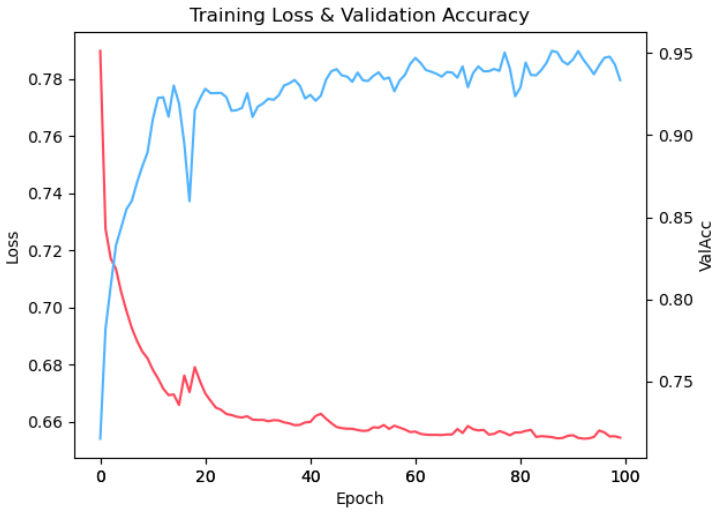


图 6: Cora 数据集链路预测任务

最终在测试集上的结果为 94.4%。

6.2 Citeseer 数据集结果

采用自环，不使用 DropEdge，PairNorm，激活函数为 relu，卷积层层数为 2 层，学习率为 0.01，隐藏层维数取 128，node_embed 维数为 64，设置误差函数为 binary_cross_entropy_with_logits 函数，设置优化器的 weigut_decay 为 5e-4，epoch 数为 100，误差变化以及在验证集上的精度变化如下所示：

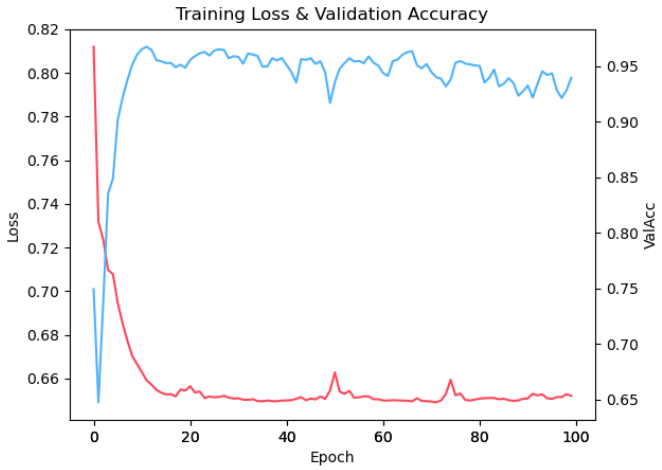


图 7: Citeseer 数据集链路预测任务

最终在测试集上的结果为 94.16%。

6.3 PPI 数据集结果

采用自环，取 PairNorm 为'PN-SI'，激活函数为 relu，设置层数为 2 层，学习率为 0.006，隐藏层维数取 128 层，设置 node_embed 维数为 64，设置误差函数为 binary_cross_entropy_with_logits 函数，不设置优化器的 weigut_decay，epoch 数为 1000，误差变化以及在验证集上的精度变化如下所示：

图 8: PPI 数据集链路预测任务

最终在测试集上的结果为 87.67%。

七、分析自环，DropEdge，PairNorm 等因素对模型的影响

由于需要分析的因素过多，因此我们就以 Cora 数据集在节点分类上的结果为例说明这些因素的影响。在上方的设置为采用自环，不使用 DropEdge，PairNorm，激活函数取得为 relu，设置卷积层层数为 2 层，学习率为 0.01，隐藏层维数取 16 层，设置误差函数为交叉熵函数，设置优化器的 weigut_decay 为 5e-4，epoch 数为 500。

7.1 自环的影响

如果不采用自环，其余设置均与上面 Cora 数据集在节点分类上的参数一致，则误差变化以及在验证集上的精度变化如下所示：

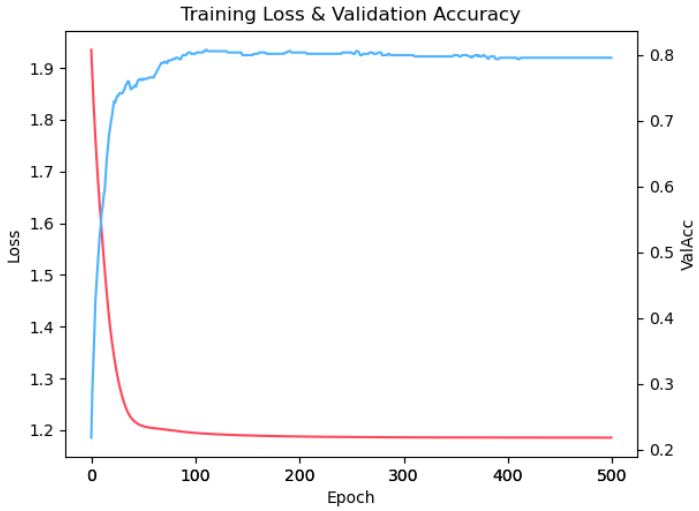


图 9: 不添加自环

最终在测试集上的结果精度为 80.7%，比起原先的 83.1% 略有降低。

7.2 层数的影响

如果采用自环，其余设置均与上面 Cora 数据集在节点分类上的参数一致，不断调整卷积层的层数，则在测试集上的精度变化如下所示：

卷积层层数	2	4	6	8	16
accuracy(%)	83.1	79.3	53.4	28.6	42.5

表 1: 实验中测试集上精度随卷积层层数的变化

可以发现当层数大于 4 的时候，层数会极大的影响整个 GCN 模型的判断能力。下面我们就在 8 层的基础上使用 DropEdge 与 PairNorm 进行试验。

7.3 DropEdge 与 PairNorm 的影响

在上一个测试中层数取 16 时的误差变化以及在验证集上的精度变化如下所示：

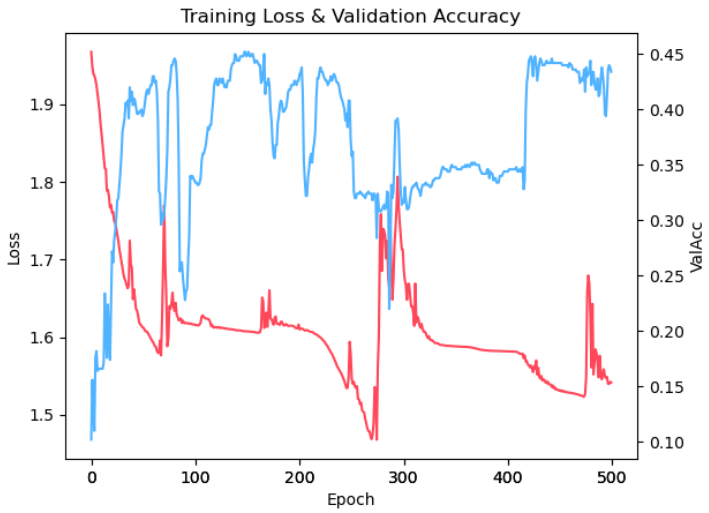


图 10: 层数为 16

如果我们此时开启 DropEdge，并设置概率为 0.3，此时得到的结果为：

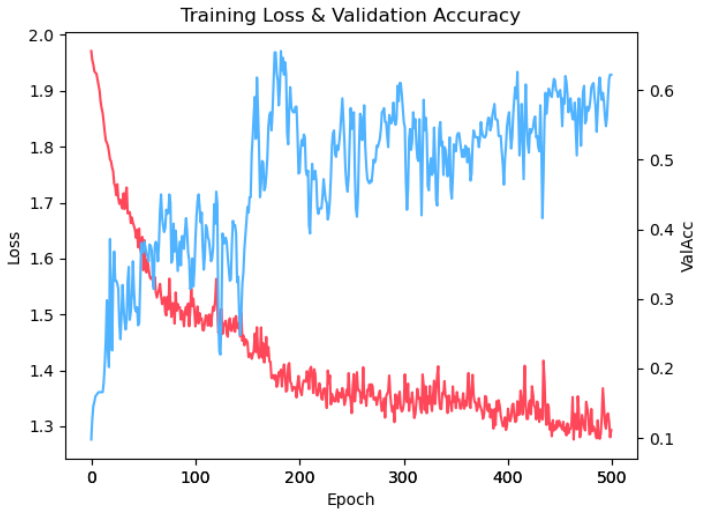


图 11: dropedge 概率为 0.3 时

此时在测试集上的概率从原先的 42.5% 变成了 60.9%，稳定性得到了一定程度的提升，如果我们不开启 Dropedge，只开启 PairNorm，则得到的结果如下所示：

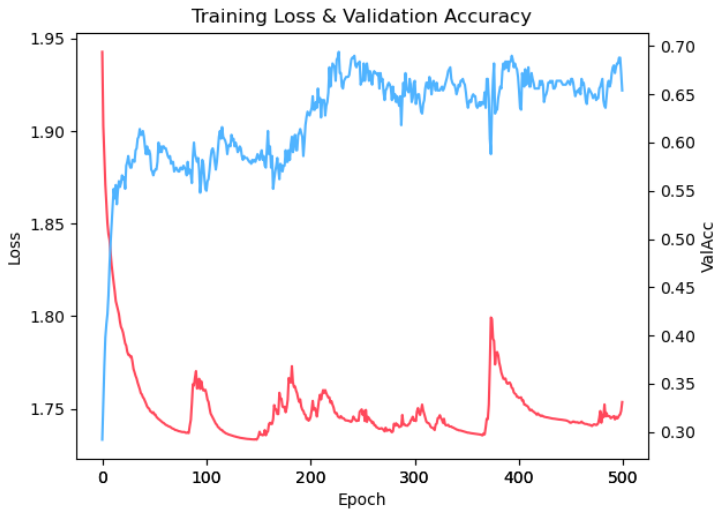


图 12: pairnorm 取”PN-SI” 时

此时在测试集上的精度为 67%。如果我们同时开启 dropedge 和 pairnorm，则得到结果如下：

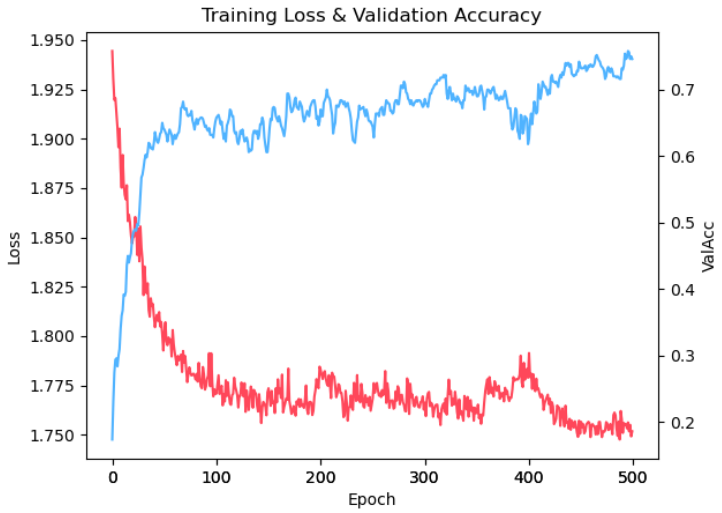


图 13: 同时开启 dropedge 和 pairnorm

此时在测试集上的结果为 75.6%，不仅在稳定性上，同时在准确率上都得到了提升。

7.4 激活函数对结果的影响

在上面的测试中激活函数取得为 relu，我们依然取层数为 2，不开启 dropedge，pairnorm 的情况，并修改激活函数观察最终结果。

最终的在测试集上的结果如下所示：

激活函数	relu	sigmoid	leaky_relu	tanh
accuracy(%)	83.1	83.1	83	83.2

表 2: 实验中测试集上精度随激活函数的变化

可以发现激活函数对 Cora 数据集的分类影响并不大，下面我们考虑采用自环，使用 DropEdge 的概率为 0.05，取 PairNorm 的形式为'PN-SI'，设置层数为 2 层，学习率为 0.006，隐藏层维数取 512 层，设置误差函数为 BCELoss 函数，不设置优化器的 weigut_decay，epoch 数为 200 时的 PPI 数据在不同激活函数下的结果（和之前设置的 PPI 节点分类不同之处在 epoch 数从 1000 降为 200 了，所以即使取 relu 的结果与上面也差距较大）：

激活函数	relu	sigmoid	leaky_relu	tanh
accuracy(%)	62.12	44.6	61.45	46.4

表 3: 实验中测试集上精度随激活函数的变化

可以发现使用 relu 或者 leaky_relu 对于结果的影响不大，但是如果换成 sigmoid 或者 tanh 就会导致无法得到很高的精度。

八、实验结论

在本次实验中，通过自己写 GCN 的卷积核使得我们对于图卷积网络有了更深刻的认识，同时也通过对节点分类以及链路预测任务的实现，加深了对图卷积网络可以使用范围的认识。