Lab 03 Report

实验内容

使用 pytorch 编写图卷积神经网络模型 GCN， 在图结构数据集 Cora 上完成如下任务:

- 节点分类： 预测节点的类别或标签。
- 链路预测： 预测两节点之间是否存在潜在的链接(边)

并研究自环、层数、DropEdge、PairNorm、
激活函数等因素对节点分类和链路预测性能的影响

实验原理

基本结构

简单来说，图卷积网络的传播方式为：

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

其中，

- $\tilde{A}$：图的邻接矩阵或邻接矩阵+自环
- $\tilde{D}$：$\tilde{A}$的度矩阵，$\tilde{D} = diag(\{\sum_j A_{ij}\}_{i=1}^{n})$
- $H^{(l)}$: 第l层的特征。对于第一层，假设其特征为X.
- $W^{(l)}$: 第l层的权重.
- σ: 激活函数

特别的，我们考虑如下的GCN：

$$f(X, \hat{A}, W) = \mathrm{softmax}\Big(\hat{A}\mathrm{ReLU}\Big(\hat{A}XW^{(0)}\Big)W^{(1)}\Big)$$

其中，$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$

节点分类：

假设节点集V被分训练集V_1，V_2. 已知V_1的标签y_1.

我们通过优化损失函数

$$\mathrm{loss} = \mathrm{nllloss}(f(X, \tilde{A}, W)|_{V1}, y_1)$$

更新权重.模型学习完成后，即可利用$f(X, \tilde{A}, W)|_{V_2}$对$V_2$进行分类预测。

链路预测

链路预测任务主要是两部分：

其一是编码（encode），它与我们前面介绍的生成节点表征是一样的(利用CGN)；

其二是解码（decode），边两端节点的表征生成边为真的几率（odds）。特别地，我们可以直接通过两个端点表征的内积表示节点间链路存在的概率。

除此之外，我们还要考虑在测试集中生成正边、负边（不存在的边，其标签为0）进行训练，以达到更好的泛化性能。

网络结构的优化

GCN存在过平滑问题，当堆叠的层数过高时，频率信息会逐渐丢失，最终导致所有节点的向量表示趋于
一致，无法区分不同节点。
为了解决过平滑问题，可以采用以下方法：

- DropEdge：随机丢弃一部分边，稀疏化邻接矩阵，减少信息的传播范围。
- PairNorm：对输出结果进行中心化和拉伸，保持节点特征的多样性。
  这些方法可以帮助在GCN模型中保留更多的局部信息，防止过度平滑化

实验步骤

1.构造卷积核

我们基于 `torch.nn` 的 `Module` ， 构造了一个GCN的卷积核

```
class GCNLayer(nn.Module):
    def __init__(self, in_channels, out_channels ):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_channels, out_channels)

    def forward(self, x, edge_index):
        row, col = edge_index
        deg = torch.bincount(row)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        x = self.linear(x)
        sp = torch.sparse.FloatTensor(edge_index, norm)
        x = torch.matmul(sp, x)
        return x
```

在构造过程中，我们利用了$\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$的稀疏性进行稀疏矩阵乘法。

2.节点预测

- **数据集划分**：利用数据集自带的属性 `train_mask` ， `val_mask` ， `test_mask` 掩码对数据集进行划分
- **GCN的构建**：

$$f(X,\hat{A},W)=\mathrm{softmax}\Big(\hat{A}\mathrm{ReLU}\Big(\hat{A}XW^{(0)}\Big)W^{(1)}\Big)$$

- **训练**：

```
def train_node_classification(model = model, optimizer = optimizer, criterion = criterion):
    model.train()
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()
```

- **验证/测试**：

通过 `val_node_classification` 计算损失：

```
@torch.no_grad()
def val_node_classification(model = model, criterion = criterion):
    model.eval()
    out = model(data.x, data.edge_index)
    loss = criterion(out[data.val_mask], data.y[data.val_mask])
    return loss.item()
```

并得到预测正确率：

```
pred = model(data.x,data.edge_index).argmax(dim=1)
correct = (pred[data.val_mask] == data.y[data.val_mask]).sum().item()
acc = int(correct) / int(data.val_mask.sum())
print(acc)
```

最后可视化训练损失，并根据验证集上的损失/正确率调参。

3.链路预测

- **边划分及负例生成**：
    - 用 `train_test_split_edges` 函数划分边：
      训练集train、验证集validation、测试集test三个部分。该函数返回测试集和验证集、测试集的正/负链路样本索引列表，以及训练集的负链路mask矩阵。
    - 为了类平衡，在每一个epoch的训练过程中，我们只需要用到与正样本一样数量的负样本。综合以上两点原因，我们在每一个epoch的训练过程中都采样与正样本数量一样的负样本，这样我们既做到了类平衡，又增加了训练负样本的丰富性。

- 需要注意，利用该方法无法获取训练集的负样本，这是因为：在该函数中进行训练集的负样本采样相对于利用了测试集的正样本信息，而作为训练集我们应该只见训练集的信息，对验证集与测试集都是不可见的。所以，在训练过程中，我们需要利用库函数negative_sampling函数仅基于训练集进行负样本采样（也就是”第二次“负采样）

- **GCN的构建**：
  与节点分类任务不同的是，我们再链路预测中的GCN不仅需要编码，还需要解码（预测边的存在概率）。我们的实现如下：

```
class Net2(torch.nn.Module):
  def __init__(self, in_channels, out_channels):
      super(Net2, self).__init__()
      self.conv1 = GCNLayer(in_channels, 128)
      self.conv2 = GCNLayer(128, out_channels)

  def encode(self, x, edge_index):
      x = self.conv1(x, edge_index)
      x = x.relu()
      return self.conv2(x, edge_index)
  #节点编码成out_channels向量

  def decode(self, z, pos_edge_index, neg_edge_index):
      edge_index = torch.cat([pos_edge_index, neg_edge_index], dim=-1)
      return (z[edge_index[0]] * z[edge_index[1]]).sum(dim=-1)  #element-wise乘法

  def decode_all(self, z):
      prob_adj = z @ z.t()
      return (prob_adj > 0).nonzero(as_tuple=False).t()

  def forward(self, x, pos_edge_index, neg_edge_index):
      return self.decode(self.encode(x, pos_edge_index), pos_edge_index, neg_edge_index)
```

`encoder` 用双层的GCN神经网络实现的， `decoder` 通过计算两端点内积预测概率。

- **训练和测试模型**：
  - 优化器：Adam
  - 损失函数：binary_cross_entropy_with_logits
  - 评价指标：AUC（on val set）
  - AUC 可视化

5. 参数调试

研究自环、层数、DropEdge、PairNorm、激活函数等因素对节点分类和链路预测性能的影响。具体结果见**实验结果**部分。

实验结果

1.调参结果

对于节点分类任务而言，经过足够多的训练，训练集上的损失都可以降低到很小，而验证集上的误差却多在75%左右。我们调参的主要目的在于增强模型的泛化性能。

对于链路预测而言，训练集上的损失很大，我们的目标在于降低训练损失。

我们在这一节中展示了部分的调参结果（验证集上）。

epoch num

迭代次数并不是越多越好。我们根据模型在验证集上的表现，选取了最佳的迭代次数。（大多在100-300）

层数

节点分类：

depth of GCN	accuracy
2	78%
3	77.2%
4	72%

链路预测

depth of GCN	AUC
2	0.91
3	0.88

自环

自环	accuracy
True	77.2%
False	78%

Edge-Drop

节点分类：

Drop Edge	accuracy
True	77%
False	78%

链路预测

Drop Edge	AUC
True	0.87
False	0.91

Pair-norm

节点分类：

Pair-norm	accuracy
True	77.4%
False	78%

链路预测

Pair-norm	AUC
True	0.91
False	0.88

激活函数

链路预测

激活函数	AUC
relu	0.91
sigmoid	0.88

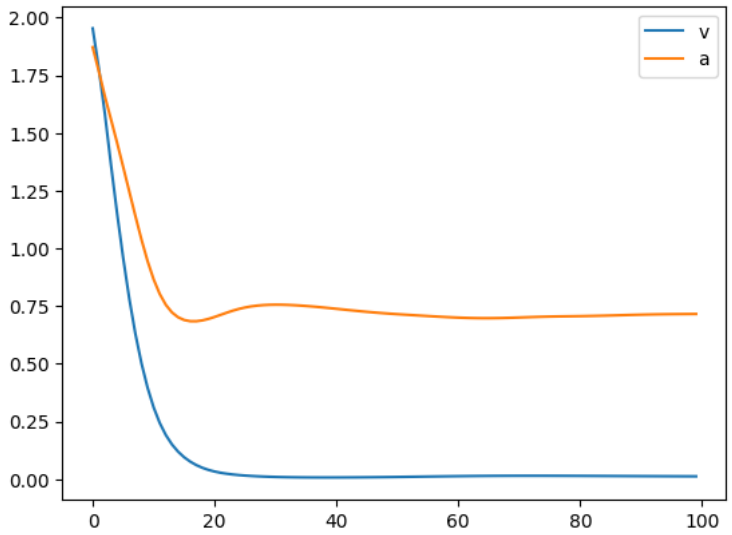
（其他调节过于繁琐，仅展示部分）

2.损失及准确率

节点分类

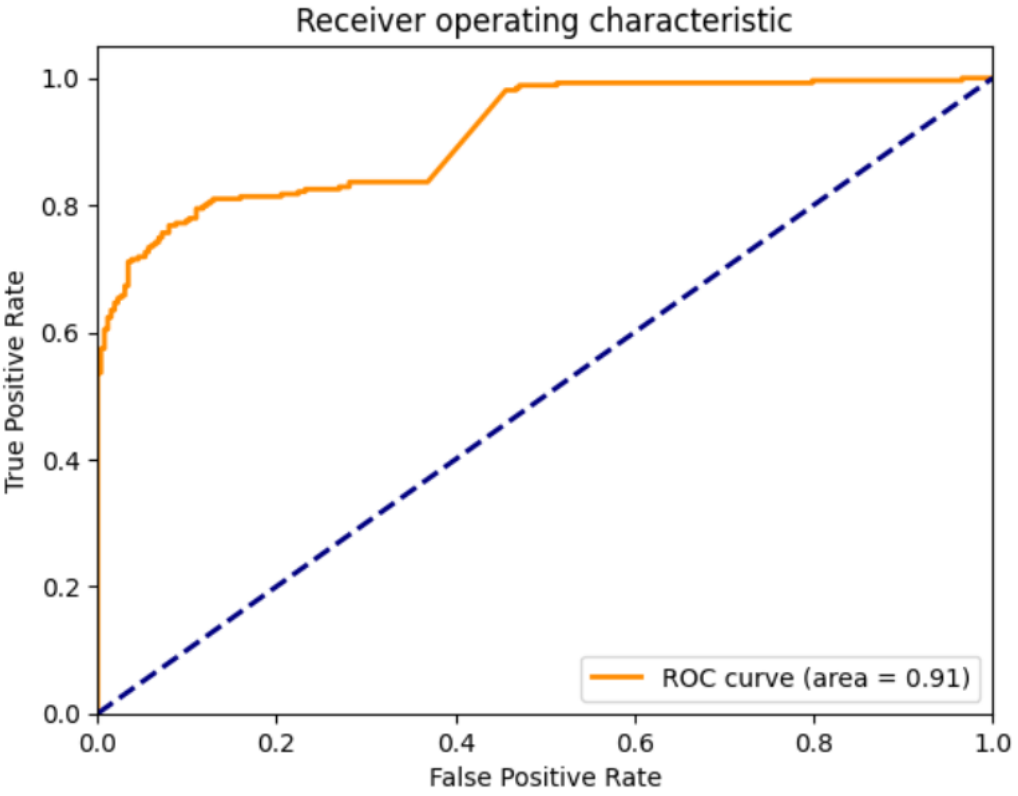
- 训练集nll损失：0.01
- 验证集交叉熵损失：0.71
- 验证集准确率：78%
- 测试集准确率：79.8%

- 训练过程损失函数图像：

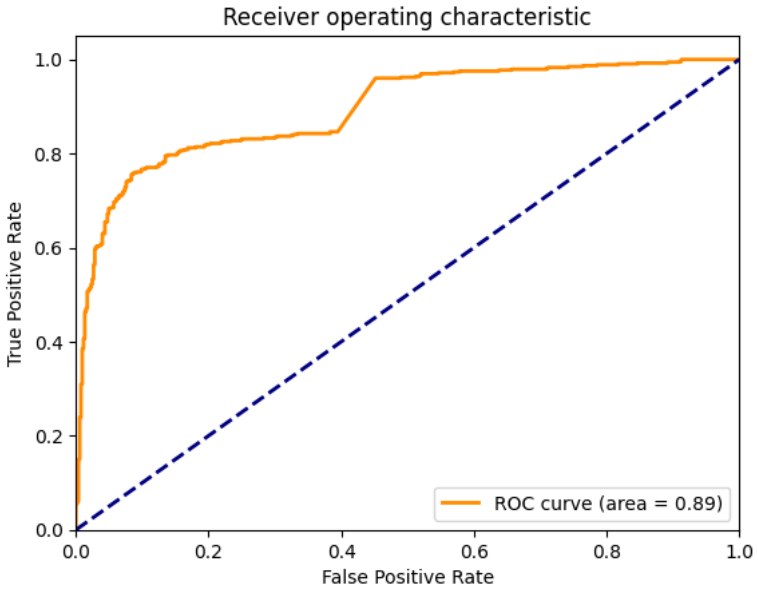


链路预测

- 训练集损失：0.39
- 验证集ROC：



- 测试集ROC：



结果分析

- 参数调节
  - 在本次任务中，两层的卷积层是较为合适的。当堆叠的层数过高时，频率信息会逐渐丢失，最终导致所有节点的向量表示趋于 一致，无法区分不同节点。
  - edge-drop、PairNorm和自环对实验影响不大，对于节点分类任务而言，几乎没有差别。但是对于链路预测任务来说，有Pairnorm的结果略好。
  - 在学习率调试过程中发现，对于某些任务而言（在本实验中链路预测），若训练集的损失是不稳定的，合理降低学习率可能会提高模型优化效果，不仅能让损失的降低更加平缓稳定，也能提高验证集上的正确率。
  - 激活函数：激活函数对实验影响很显著。Relu函数的效果是明显优于Sigmoid的。
- 泛化性能的提高

  在训练集上，损失值低，正确率通常可以很高，但是在验证集上，损失却难以下降，正确率也几乎在75%左右波动，我们尝试了一些提高泛化能力的方法：比如说减少迭代次数、dropout等。但是效果均不显著。

  可能需要进一步优化模型结构，或者进行更好的特征工程优化数据，方可提高模型的性能。