深度学习 实验3 利用图卷积神经网络完成节点分类和链路预测

> 本次报告详细的超参调试和分析以数据集 'cora' 为主， 'citeseer' 数据集的调试雷同，简要列出了具体参数设置。

1 节点分类

1.1 数据生成

以通过 PyTorch Geometric 库中的 Planetoid 数据集获取，并对数据集进行预处理。

```
# 加载Cora数据集
cora = Planetoid(
    ".",
    "Cora",
    split='geom-gcn',
    transform=RandomNodeSplit( # 随机划分数据集
        num_val=300,
        num_test=500,
    )
)
```

1.2 模型搭建

大纲

图卷积层（`GraphConv`）

- **功能**：实现了图卷积操作，其中自环是明确添加到每个节点上的，以确保节点能够考虑到自身的特征。
- **主要操作**：节点特征通过线性变换，并结合自环和归一化的邻接矩阵进行图卷积。

归一化层（`PairNorm`）

- **功能**：提供对输入特征的成对归一化，帮助模型在训练过程中保持稳定性，减少内部协变量偏移。
- **主要操作**：计算输入特征的均值和方差，并使用这些统计量来归一化数据，再通过一个可调整的尺度因子进行缩放。

GCN模型（`GCN`）

- **功能**：构建一个用于节点分类的图卷积网络，包含三个图卷积层和一个Softmax分类层。
- **主要操作**：每个图卷积层后使用`Tanh`激活函数，可选的`PairNorm`归一化（实际未使用），最终通过Softmax层输出分类概率。

训练函数（`train`）

- **功能**：执行模型的一个训练周期，包括前向传播、损失计算、反向传播、参数更新。
- **主要操作**：计算训练和验证损失，并更新模型参数。同时计算验证集上的准确率作为模型性能的一个指标。

图卷积层：
基于PyTorch框架，其主要功能是将输入的节点特征矩阵`x`和边索引矩阵`edge_index`进行图卷积操作，并输出新的节点特征矩阵。

1. **添加自环**：通过`add_self_loops`函数向图中的每个节点添加自环。
2. **计算度矩阵**：利用边索引计算图中每个节点的度，并进一步求得度矩阵的逆平方根`deg_inv_sqrt`。这是为了后续进行特征的归一化处理，确保信息传播过程中各节点的影响力被均衡处理。
3. **归一化处理**：使用节点的度的逆平方根对边权重进行归一化，得到归一化后的邻接矩阵`adj`。为了防止节点度数过大导致的梯度爆炸问题。
4. **线性变换**：通过`nn.Linear`模块对节点特征进行线性变换，以学习特征间的非线性关系和提升模型的表达能力。
5. **图卷积操作**：将线性变换后的节点特征与归一化的邻接矩阵相乘，实现图卷积。

```
# 定义图卷积层
class GraphConv(nn.Module):
    """
    Defines a Graph Convolutional Layer.
    """
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.linear = nn.Linear(in_channels, out_channels)

    def forward(self, x: torch.Tensor, edge_index: torch.Tensor) →
torch.Tensor:
        # Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Compute the degree matrix
        row, col = edge_index
        deg = self.degree(row, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]
```

```
        # inplace inf by 0
        norm[torch.isinf(norm)] = 0
        # Compute the adjacency matrix
        adj = torch.sparse_coo_tensor(edge_index, norm, (x.size(0),
x.size(0)), dtype=x.dtype, device=x.device)

        # Perform the linear transformation
        x = self.linear(x)

        # Perform the convolution
        return adj @ x

    @staticmethod # 静态方法
    def degree(index: torch.Tensor, num_nodes: int, dtype: torch.dtype
= None) -> torch.Tensor:
        """
        Compute the degree of each node.
        """
        out = torch.zeros(num_nodes, dtype=dtype, device=index.device)
        out.scatter_add_(0, index, out.new_ones((index.size(0),)))
        return out
```

图卷积网络：
该模型通过堆叠三个图卷积层和一个Softmax分类层来处理图数据，使用Tanh作为激活函数。

关于参数设置：

1. **输入节点特征的维度**：由 `cora.num_features` 指定，即输入到第一个图卷积层 `conv1` 的节点特征维度。
2. **图卷积层数**：初始设置使用三个图卷积层，经过调参保留了这个设置。
3. **PairNorm**：代码中包括了 `PairNorm` 层的初始化，前向传播函数中默认不使用 `PairNorm`。
4. **激活函数类型**：初始默认为 `ReLU`，经过调参，使用了 `Tanh` 激活函数。
5. **是否为每个节点添加自环边**：默认为是，详见图卷积层的定义。
6. **边丢弃率**：初始设置为0.1，在调参后确定为0.04（调用代码见train）。

```
# 定义GCN模型，堆叠三个图卷积层和一个Softmax分类层，使用Tanh激活函数
class GCN(nn.Module):
    """
    Defines the Graph Convolutional Network (GCN) for node
classification.
    """
    def __init__(self):
```

```
        super().__init__()
        # First graph convolutional layer with input and output
dimensions
        self.conv1 = GraphConv(cora.num_features, 500)
        self.norm = PairNorm()

        # Second graph convolutional layer
        self.conv2 = GraphConv(500, 100)
        # Third graph convolutional layer mapping to the number of
classes
        self.conv3 = GraphConv(100, cora.num_classes)
        # Softmax classifier for the final layer
        self.classifier = Softmax(dim=1)

    def forward(self, x: torch.Tensor, edge_index: torch.Tensor) ->
torch.Tensor:
        """
        Forward propagation logic.
        """
        # First Convolutional Layer followed by Tanh activation and
PairNorm
        # x = self.norm(x)
        h = self.conv1(x, edge_index)
        h= h.tanh()
        # h = self.norm(h)

        # Second Convolutional Layer followed by Tanh activation and
PairNorm
        h = self.conv2(h,edge_index)
        h = h.tanh()
        # h = self.norm(h)

        # Third Convolutional Layer followed by Tanh activation and
PairNorm
        h = self.conv3(h,edge_index)
        h=h.tanh()

        # Apply the softmax classifier
        h = self.classifier(h)

        return h

def drop_edge(edge_index, drop_prob): # 随机丢弃边（正则化）

    if drop_prob < 0 or drop_prob > 1:
        raise ValueError("`drop_prob` must be between 0 and 1")
```

```
    # Calculate the number of edges to drop
    num_edges = edge_index.size(1)
    num_edges_drop = int(drop_prob * num_edges)

    # Generate a mask for dropping edges
    mask = torch.ones(num_edges, dtype=torch.bool)
    drop_indices = torch.randperm(num_edges)[:num_edges_drop]
    mask[drop_indices] = False

    return edge_index[:, mask]
```

1.3 模型训练

定义训练函数：
先加载和分割数据集，然后初始化GCN模型，选择Adam优化器，并设定学习率=0.01，使用交叉熵损失函数来进行节点分类的训练。
早停：如果验证损失没有显著改善，增加耐心值并在连续五个周期无改进时停止训练，防止过拟合。

```
def train_cora(test_pc: float, epochs: int) -> tuple:
    num_test = int(cora[0].num_nodes * test_pc)
    num_val = int(cora[0].num_nodes * 0.1) # 验证集比例

    # 重新加载数据集
    dataset = Planetoid(
        ".",
        "Cora",
        split='geom-gcn',
        transform=RandomNodeSplit(
            num_val=num_val,
            num_test=num_test,
        )
    )
    data = dataset[0]

    # 初始化模型、优化器和损失函数
    model = GCN()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    criterion = torch.nn.CrossEntropyLoss()

    train_loss_arr = []
    val_loss_arr = []
    test_loss_arr = []
    train_acc_arr = []
    val_acc_arr = []
```

```
    test_acc_arr = []

    # 早停
    patience = 0
    epsilon = 0.0001
    min_val_loss = float('inf')  # 确保第一个epoch一定会更新

    for epoch in range(epochs):
        raw_edge_index = data.edge_index
        data.edge_index = drop_edge(data.edge_index, 0.04) # dropedge,
设置为0.04
        train_loss, val_loss, val_acc = train(model, optimizer,
criterion, data)

        train_loss_arr.append(train_loss.detach().numpy())

        # 早停
        if epoch == 0 or (min_val_loss - val_loss) > epsilon:
            min_val_loss = val_loss
            patience = 0
        else:
            patience += 1
            print(f'Early stopping patience counter: {patience}') # 早
停标记
            if patience ≥ 5:
                print(f'Early stopping triggered at epoch {epoch}')
                break

        val_loss_arr.append(val_loss.detach().numpy())

        # 计算训练集准确率
        train_acc = eval(model, data, data.train_mask)
        train_acc_arr.append(train_acc)

        # 计算验证集准确率
        val_acc_arr.append(val_acc)

        # 计算测试集loss
        test_loss = criterion(model(data.x, data.edge_index)
[data.test_mask], data.y[data.test_mask])
        test_loss_arr.append(test_loss.detach().numpy())
        test_acc = eval(model, data, data.test_mask)
        test_acc_arr.append(test_acc)

        print(f'Epoch: {epoch} \t Training loss: {train_loss} \t
Validation loss: {val_loss} \t Validation Accuracy: {val_acc}')
        data.edge_index = raw_edge_index # 恢复原始边索引
```

```
    # 最终在测试集上评估模型
    test_acc = eval(model, data, data.test_mask)
    test_cf = compute_confusion_matrix(model, data)

    return train_loss_arr, val_loss_arr, test_loss_arr, train_acc_arr,
val_acc_arr, test_acc_arr, test_acc, test_cf
```

1.4 调参分析

针对以下参数：**自环、层数、DropEdge 、PairNorm 、激活函数**进行修改与验证集上的考量，以获得最佳表现的参数组合。

默认值：

```
自环=true,
层数=3,
dropedge=0.1,
pairnorm=false,
激活函数=relu.
```

调参的思路为，先确定大框架，比如网络层数以及激活函数，之后再对有优化效果的参数，如是否设置自环，pairnorm, dropedge, 进行调整。

首先确定网络层数。默认值为3，在其他参数不变的情况下，2层和3层网络的验证准确率都在74%左右。而再减或者再加都会导致验证准确率下降（梯度消失或爆炸）。考虑到其他参数的调整可能会消除层数增多带来的准确率下降，为保证训练效果，选择网络层数为3.

在网络深度比较低的时候，relu, leaky_relu 和 tanh 激活函数的表现性能都差不多，sigmoid 激活函数的表现性能很差；而当网络深度高的时候，relu 和 leaky_relu 激活函数的表现性能都差不多，然后tanh 的表现性能其次，然后 sigmoid 激活函数的表现性能依然很差。由于选择了3层网络，因此选择tanh作为激活函数（因为它比relu的训练准确率略高）。

接下来考虑自环。添加自环后，归一化后的拉普拉斯矩阵的最大特征值会变小，因此会在一定程度上缓解梯度更新出现的问题。所以在深层网络下，加自环的效果比较好。
同时图卷积中加自环是为了更好地利用节点自身的信息，从而提高模型的表现。在图卷积中，节点的特征是由其自身特征和邻居节点特征加权求和得到的。如果不考虑节点自身特征，那么模型就无法利用这一重要信息。并且在一些情况下，加入自环可以提高模型的稳定性，避免模型过于依赖邻居的节点特征，从而提高模型的泛化能力。取消自环后发现验证准确率的确有所下降，因此保留自环的添加。

dropedge是一种随机丢弃边的技术，可以帮助模型减少过拟合和提高泛化能力。较低的丢弃率通常会产生更好的性能和更高的稳定性。如果丢弃率过高，可能会导致信息损失和不稳定的训练过程。经过调试，DropEdge 可以改善模型的过拟合现象，但是在本次实验中，DropEdge 设置的过高并没有明显的体现出对模型泛化能力的提高。原设置0.1的丢弃率过高，在下降到0.04时模型表现较好。

pairnorm更适合使用在层数更深的网络中。图卷积神经网络中，不同节点的度数可能会差异较大，这会导致在网络层之间传递的信息被不同的尺度所影响，从而导致梯度消失或爆炸的问题。PairNorm 技术通过对每对节点之间的特征进行归一化，使得每个节点在计算过程中受到相同的尺度影响，从而缓解了深层网络中的梯度问题，以获得更好的性能和更高的稳定性。而在低层数的网咯中，比如本次实验选取的3层网络中，梯度传递问题的影响相对较小，此时该技术对于缓解梯度问题的作用相对较小，甚至引入一些额外的噪声或不稳定性，因此严重影响了模型的性能表现。故不采取。

1.5 模型测试

在确定参数为：

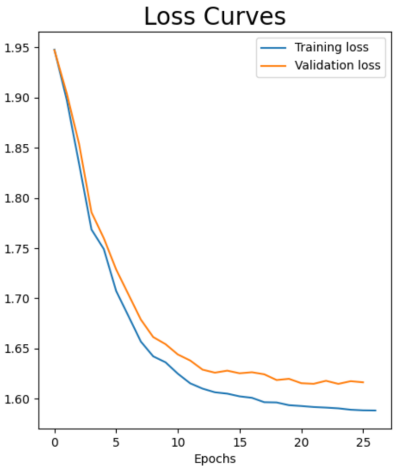
```
自环=true,
层数=3,
dropedge=0.04,
pairnorm=false,
激活函数=tanh
```

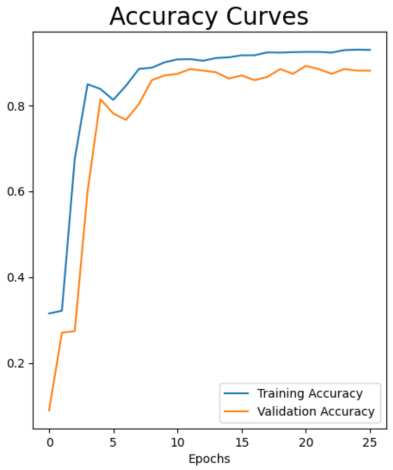
后，在测试集上进行测试(epoch = 100，但是均发生了早停)：

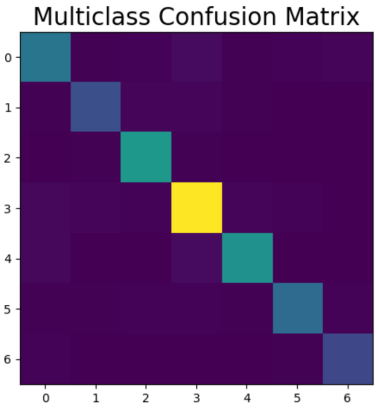
`train_loss_arr, val_loss_arr,val_acc, test_acc, test_cf = train_cora(0.2, 100)`

得到结果：

```
test_acc: 0.8613678216934204
```







cite_seer数据集的参数设置与cora一致，需要指出的不同是自环的添加。
在Citeseer数据集的程序中，自环是在初始化GCN模型时添加的：

1. **计算邻接矩阵**：首先通过 `edge_index_to_adjacency_matrix` 函数将边索引转换成邻接矩阵。
2. **添加自环**：通过向邻接矩阵加上单位矩阵 `torch.eye(adj.size(0))` 来实现自环的添加。这样做的效果是，每个节点都与自身直接相连，邻接矩阵的对角线元素都被设置为1。
3. **归一化处理**：添加自环后，对邻接矩阵进行对称归一化处理，计算方式是 $D^{-1/2}AD^{-1/2}$，其中D是度矩阵的对角阵，A是添加了自环的邻接矩阵。

这种方式的优点是在模型初始化阶段就一次性处理好了自环添加和邻接矩阵的归一化，便于后续的图卷积操作直接使用处理好的邻接矩阵。

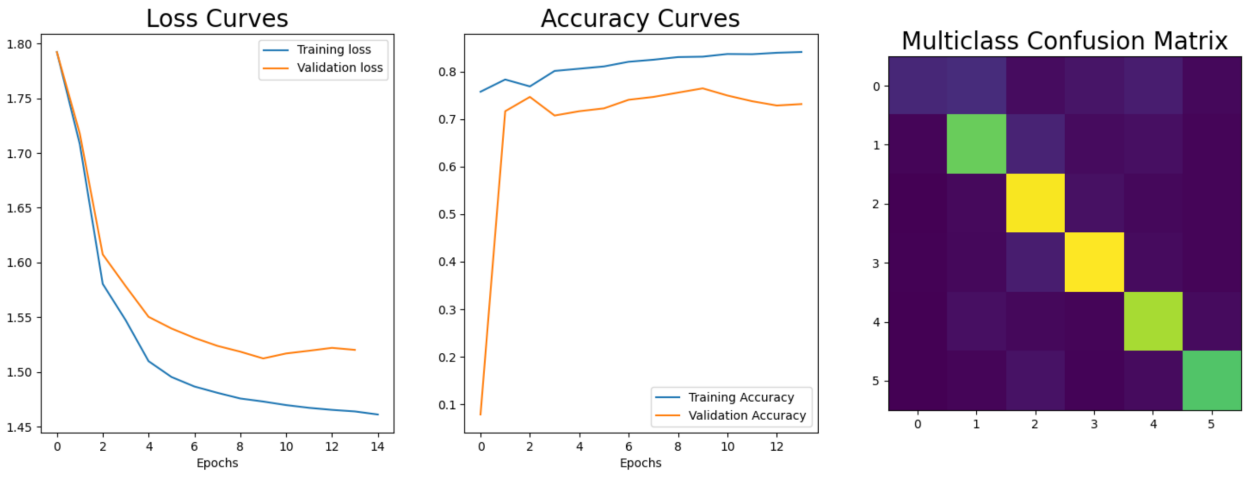
在Cora数据集的程序中，自环的添加是在每次图卷积层的前向传播时动态完成的：

1. **图卷积层实现**：每个 `GraphConv` 层在其前向传播函数 `forward` 中处理自环的添加。
2. **动态添加自环**：使用 `add_self_loops` 函数直接在每次前向传播时将自环添加到当前的边索引 `edge_index` 中。这个函数将边索引和节点总数作为输入，输出添加了自环的新边索引。
3. **度矩阵计算**：随后根据新的边索引计算度矩阵，并进行归一化处理，和Citeseer中的处理方式类似。

这种方式的优点是灵活性更高，可以在每次前向传播中根据需要动态调整自环的添加。这对于一些动态图或者在训练过程中需要调整图结构的场景比较有用。

cite_seer数据集的结果：

```
test_acc: 0.7338345646858215
```



2 链路预测

2.1 数据生成和处理

数据加载与预处理步骤：

1. **数据加载**：
   - 使用 `torch_geometric.datasets.Planetoid` 从 `PyTorch Geometric` 库中加载Cora数据集。

```
cora = Planetoid(".", "Cora", split='geom-gcn')
data = cora[0]
```

2. **划分数据集**：
   - 将数据集划分为训练集、验证集和测试集。
   - 利用 `train_test_split` 方法将边索引随机分为训练集、验证集和测试集，并生成对应的掩码。

```
num_edges = 2 * data.edge_index.size(1)
edge_indices = np.arange(num_edges)
train_indices, rest_indices = train_test_split(edge_indices,
test_size=0.3, random_state=42)
val_indices, test_indices = train_test_split(rest_indices,
test_size=0.5, random_state=42)
train_mask = torch.zeros(num_edges, dtype=torch.bool)
val_mask = torch.zeros(num_edges, dtype=torch.bool)
test_mask = torch.zeros(num_edges, dtype=torch.bool)
train_mask[train_indices] = True
val_mask[val_indices] = True
test_mask[test_indices] = True
data.train_mask = train_mask
data.val_mask = val_mask
data.test_mask = test_mask
```

2.2 模型搭建

本实验使用图卷积网络（GCN）进行节点分类和链接预测。

1. **图卷积层（GCNConv）**：
   - 同节点分类。

```
class GCNConv(nn.Module):
    def __init__(self, in_features, out_features):
        super(GCNConv, self).__init__()
```

```
        self.weight = nn.Parameter(torch.FloatTensor(in_features,
out_features))
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.xavier_uniform_(self.weight)

    def forward(self, input, adj):
        support = torch.mm(input, self.weight)
        output = torch.spmm(adj, support)  # Sparse matrix
multiplication
        return output
```

2. **PairNorm层**：
   - 用于对节点特征进行归一化，减小不同节点之间特征的差异。

```
class PairNorm(nn.Module):
    def __init__(self, scale: float = 1, eps: float = 1e-5):
        super().__init__()
        self.scale = scale
        self.eps = eps

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        mean = x.mean(dim=0, keepdim=True)
        var = ((x - mean) ** 2).mean(dim=0, keepdim=True)
        x = (x - mean) / torch.sqrt(var + self.eps)
        x = self.scale * x
        return x
```

3. **链接预测模型**：
   - 包含三个GCN卷积层和一个PairNorm层。
   - 使用邻接矩阵进行特征传播。

```
class LinkPrediction(nn.Module):
    def __init__(self, num_features, hidden_channels, edge_index):
        super(LinkPrediction, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)
        adj = edge_index_to_adjacency_matrix(edge_index)
        adj = adj + torch.eye(adj.size(0))
        D = torch.diag(torch.pow(adj.sum(1).float(), -0.5))
        A = torch.mm(torch.mm(D, adj), D)
        self.A = A
```

```
        self.norm = PairNorm()

    def forward(self, x, edge_index):
        x = self.norm(x)
        x = self.conv1(x, self.A)
        x = x.relu()
        x = self.norm(x)
        x = self.conv2(x, self.A)
        x = x.relu()
        x = self.norm(x)
        x = self.conv3(x, self.A)
        return x

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

2.3 模型训练

1. **定义损失函数和优化器**：
   - 使用 `BCEWithLogitsLoss` 作为损失函数，用于二分类任务。
   - 使用 `Adam` 优化器来更新模型参数。

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.BCEWithLogitsLoss()
```

2. **训练和验证步骤**：
   - 在每个epoch中，首先进行训练，然后在验证集上评估模型性能。
   - 记录每个epoch的训练损失、验证损失和验证AUC。

```
def train_link_prediction(model, data, optimizer, criterion,
num_epochs=100):
    train_loss_arr, val_loss_arr, auc_arr = [], [], []
    for epoch in range(num_epochs):
        neg_edge_index = negative_sampling(data.edge_index,
num_neg_samples=data.edge_index.size(1))
        loss = train(model, optimizer, criterion, data,
data.edge_index, neg_edge_index)
        loss2, auc = val(model, data, neg_edge_index, criterion)
        print(f'Epoch: {epoch}, Train Loss: {loss}, Val Loss:
{loss2}, Val AUC: {auc}')
```

```
        train_loss_arr.append(loss.detach().numpy())
        val_loss_arr.append(loss2.detach().numpy())
        auc_arr.append(auc)
    return train_loss_arr, val_loss_arr, auc_arr
```

3. **绘制损失曲线和AUC曲线**：
   - 训练完成后，绘制训练损失、验证损失和验证AUC的变化曲线。

```
train_loss_arr, val_loss_arr, auc_arr =
train_link_prediction(model, data, optimizer, criterion)

fig, axs = plt.subplots(1, 2, figsize=(15, 5))

axs[0].plot(train_loss_arr, label='Train Loss')
axs[0].plot(val_loss_arr, label='Validation Loss')
axs[0].set_title('Train and Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend()

axs[1].plot(auc_arr, label='Validation AUC', color='orange')
axs[1].set_title('Validation AUC over Epochs')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('AUC')
axs[1].legend()

plt.tight_layout()
plt.savefig('cora_combined.png')
```

4. **测试模型**：
   - 在测试集上评估模型性能，计算并打印测试集上的AUC值。

```
test_auc = test(model, data, negative_sampling(data.edge_index,
num_neg_samples=data.edge_index.size(1)), criterion)
print(f'Test AUC: {test_auc}')
```

2.4 调参分析

和节点分类一样，针对以下参数：**自环、层数、DropEdge 、PairNorm 、激活函数**进行修改与验证集上的考量，以获得最佳表现的参数组合。

默认值：

```
自环=true,
层数=3,
dropedge=0.1,
pairnorm=false,
激活函数=relu.
```

调参的思路依旧为，先确定大框架，比如网络层数以及激活函数，之后再对有优化效果的参数，如是否设置自环，pairnorm, dropedge, 进行调整。

首先确定网络层数。比较不同层数对图卷积网络性能的影响。随着网络深度的增加，模型的性能在一定程度上有所提高。但是当网络深度不断增加时，模型的性能不断下降，可能是因为过深的网络容易出现梯度消失或梯度爆炸的问题。
与节点分类问题不同的是，其模型性能下降的幅度并不是很大，可能是因为在链路预测任务中，每条边的标签只有两种情况，相比于节点分类任务，数据的复杂性更低。因此，在链路预测任务中，即使网络较深，模型也能够较好地学习到节点之间的关系，从而提高模型性能。此外，链路预测任务的数据规模通常比节点分类任务小得多，这也有助于缓解过拟合和梯度消失的问题。因此，与节点分类任务不同，链路预测任务中的图卷积网络可能比较容易处理深层次的网络结构。

在网络深度比较低的时候，relu, leaky_relu 和 tanh 激活函数的表现性能都差不多，sigmoid 激活函数的表现性能很差；而当网络深度高的时候，relu 和 leaky_relu 激活函数的表现性能都差不多，然后tanh 的表现性能其次，然后 sigmoid 激活函数的表现性能依然很差。由于选择了3层网络，因此选择tanh作为激活函数（因为它比relu的训练准确率略高）。

比较了不同层数下是否添加自环对模型性能的影响。经过测试，在不同网络深度的情况下，加自环的模型在训练AUC和验证AUC上均优于不加自环的模型。图卷积层中的对称化拉普拉斯矩阵的特征值会在叠加多层图卷积层操作后出现数值不稳定和梯度爆炸的问题，因此添加自环后，归一化后的拉普拉斯矩阵的最大特征值会变小，因此会在一定程度上缓解梯度更新出现的问题。所以在深层网络下，加自环的效果比较好。
同时图卷积中加自环是为了更好地利用节点自身的信息，从而提高模型的表现。在图卷积中，节点的特征是由其自身特征和邻居节点特征加权求和得到的。如果不考虑节点自身特征，那么模型就无法利用这一重要信息。并且在一些情况下，加入自环可以提高模型的稳定性，避免模型过于依赖邻居的节点特征，从而提高模型的泛化能力。
总的来说，加自环对于图卷积神经网络的性能和稳定性有着一定的积极影响。

比较在不同层数下使用不同的 DropEdge 丢弃率对图卷积网络性能的影响。从表中可以看出，随着丢弃率的增加，模型的训练损失和验证损失都有所增加，同时训练AUC和验证AUC都有所下降。这是因为丢弃边会减少网络的有效连接，从而减少了网络的学习能力，并可能导致一些关键信息的丢失。但是在一定范围内，适当的丢弃边可以防止模型过拟合，并提高模型的泛化性能。但是，如果丢弃率过高，可能会导致信息损失和不稳定的训练过程。在本次实验的调参过程种，不加 DropEdge 反而有利于提升预测准确率。

pairnorm更适合使用在层数更深的网络中。图卷积神经网络中，不同节点的度数可能会差异较大，这会导致在网络层之间传递的信息被不同的尺度所影响，从而导致梯度消失或爆炸的问题。PairNorm 技术通过对每对节点之间的特征进行归一化，使得每个节点在计算过程中受到相同的尺度影响，从而缓解了深层网络中的梯度问题，以获得更好的性能和更高的稳定性。而在低层数的网络中，比如本次实验选取的3层网络中，梯度传递问题的影响相对较小，此时该技术对于缓解梯度问题的作用相对较小，甚至引入一些额外的噪声或不稳定性，因此严重影响了模型的性能表现。
在高层数的网络中使用 PairNorm 技术可以显著提高模型的性能和稳定性。与节点分类问题一样，在低层数的网络中使用 PairNorm 效果并不是很好，而在高层数的网络中使用 PairNorm 效果比较好。

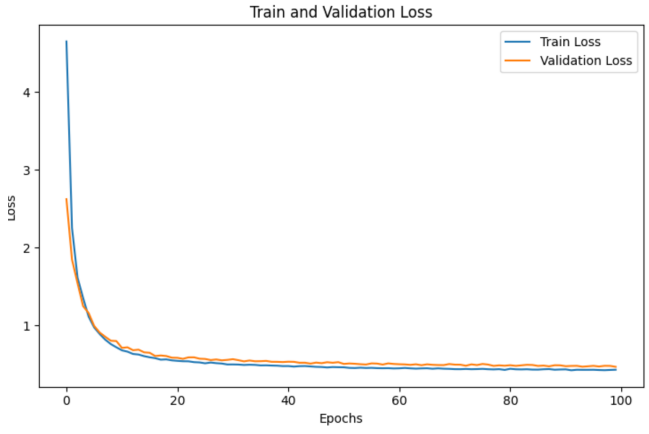
确定参数为：

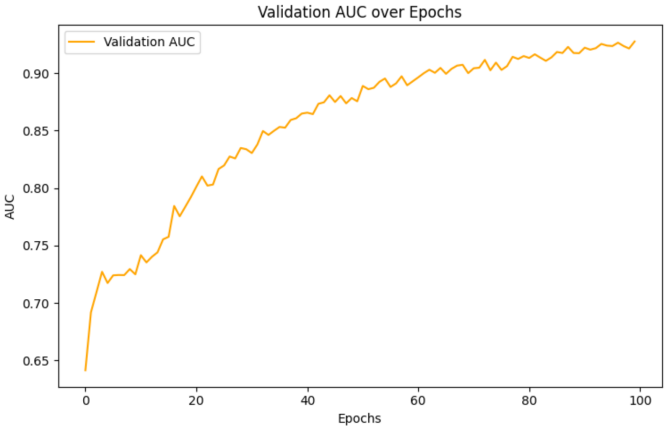
```
自环=true,
层数=3,
dropedge=0,
pairnorm=true,
激活函数=tanh
```

2.5 模型测试

在确定超参后，添加在测试集测试的代码，运行得到：

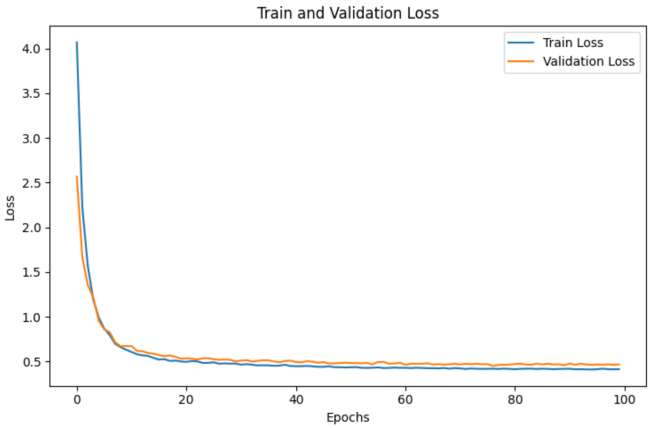
```
Test Loss: 0.48938062787055597, Test AUC: 0.9242295818847591
Test AUC: 0.9242295818847591
PS C:\Users\Miner\OneDrive\Code\python\2024\spring\deep learning lab\lab3\src>
```

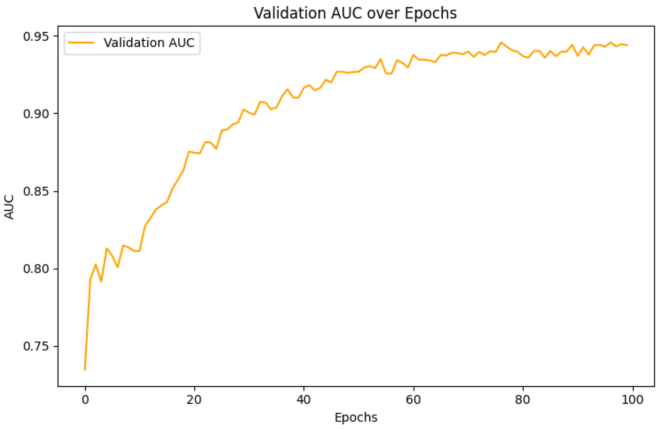




在 citeseer数据集上的运行结果如下：

```
Test Loss: 0.44738551974296570, Test AUC: 0.94600723034170440
Test AUC: 0.9460072303417044
PS C:\Users\Miner\OneDrive\Code\python\2024\spring\deep learning lab\lab3\src>
```


Train and Validation Loss
Train Loss
Validation Loss
Loss
Epochs
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100


Validation AUC over Epochs
Validation AUC
AUC
Epochs
0.95
0.90
0.85
0.80
0.75
0
20
40
60
80
100