实验四 GCN

袁雨 PB20151804

一、实验要求

使用pytorch 或者 tensorflow 的相关神经网络库编写图卷积神经网络模型 GCN， 并在相应的图结构数据集上完成节点分类和链路预测任务，最后分析自环、层数、 DropEdge 、PairNorm 、激活函数等因素对模型的分类和预测性能的影响 。

二、实验准备

1. **网络框架**

   选择了 pytorch ， 并安装了 torch_geometric。

2. **数据准备**

   本次实验使用的数据包含三个常用的图结构数据集： Cora 、Citeseer 、PPI 。

- **Cora**

  Cora数据集由机器学习论文组成。 这些论文分为以下七个类别之一：基于案例、遗传算法、神经网络、概率方法、强化学习、规则学习、理论。这些论文的选择方式是，在最终语料库中，每篇论文引用或被至少一篇其他论文引用。整个语料库中有 2708篇论文。在词干堵塞和去除词尾后，只剩下 1433个 唯一的单词。文档频率小于10的所有单词都被删除。

  该数据集由 cora.content 和 cora.cites 两个文件组成。

  - cora.content

    包含以下格式的论文描述：<paper_id> <word_attributes>+ <class_label>。

    每行（其实就是图的一个节点）的第一个字段是论文的唯一字符串标识，后跟 1433 个字段（取值为二进制值），表示1433个词汇中的每个单词在文章中是存在(由1表示)还是不存在（由0表示）。最后，该行的最后一个字段表示论文的类别标签（7个）。

  - cora.cites

    包含语料库的引用关系图。每行（其实就是图的一条边）用以下格式描述一个引用关系：<被引论文编号> <引论文编号>。

    每行包含两个paper id。第一个字段是被引用论文的标识，第二个字段代表引用的论文。引用关系的方向是从右向左。如果一行由“论文1 论文2”表示，则“论文2 引用 论文1”，即链接是“论文2 - >论文1”。

- **Citeseer**

  CiteSeer数据集包含3312篇论文，分为6类：Agents、AI、DB、IR、ML、HCI。引用网络由4732个链接组成。数据集中的每个出版物都用0/1值的词向量描述，该词向量指示字典中是否存在相应的词。该词典包含3703个独特的单词。

  该数据集也由 citeseer.content 和 citeseer.cites 两个文件组成。存储格式与 Cora 数据集类似。

- **PPI**

  PPI(生物化学结构) 网络是蛋白质相互作用（Protein-Protein Interaction,PPI）网络的简称。PPI是指两种或以上的蛋白质结合的过程，通常旨在执行其生化功能。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数据集共24张图，每张图对应不同的人体组织，平均每张图有2371个节点，共56944个节点818716条边，每个节点特征长度为50，其中包含位置基因集，基序集和免疫学特征。基因本体基作为label(总共121个)，label不是one-hot编码。

  - valid_feats.npy

    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-class_map.json

    为节点的label文件，shape为（121, 56944）,每个节点的label为121维。

  - ppi-G.json

    为节点和链接的描述信息，节点：{“test”: true, “id”: 56708, “val”: false}, 表示 id 为56708的节点是否为 test 集或者 val 集，链接：“links”: [{“source”: 0, “target”: 372}, {“source”: 0, “target”: 1101}，表示 id 为0的节点和 id 为1101的节点之间有 links。

  - ppi-walks.txt

    为链接信息。

  - ppi-id_map.json

    为节点id信息。

三、实验原理

（一）图表示学习

- 定义

  给定一个图 G=(V, E)，将图上的节点压缩成低维的向量表示$R^{n\times k}(k \ll n)$

- 性质
  - 相邻节点具有相似的向量表示
  - 具有相似属性的节点具有相似向量表示
  - 节点的顺序变换对向量表示没有影响
- 应用
  - 节点分类

    预测节点类别

  - 链路预测

    预测两个节点是否有边相连

（二）图的基本概念

- 图的矩阵表示
  - 邻接矩阵 $A_{ij}=1$, 如果 v_i 和 v_j 相邻
  - 度矩阵 $\boldsymbol{D}=\operatorname{diag}(d\left(v_1\right), \ldots, d\left(v_N\right))$

$$d\left(v_i\right)=\sum_{v_j \in \mathcal{N}_{v_i}} A_{ij}$$

- 拉普拉斯矩阵
  - 定义

    $L=D-A$
  - 性质
    - $L1=D1-A1=d-d=0$
    - $1L=0$
    - L是半正定的，最小特征值为0，其特征向量为1
  - 归一化后的拉普拉斯矩阵
    - 对称归一化: $L^{\mathrm{sym}}=D^{-\frac{1}{2}} L D^{-\frac{1}{2}}=I-D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$
    - 随机游走归一化: $L^{rw}=D^{-1} L=I-D^{-1} A$
- 图傅里叶变换
  - 图傅里叶变换
    - $L=U \Lambda U^{\top}$ 为其特征值分解， U 的列向量类比于傅里叶变换中的基
    - 对图上信号 f 的图傅里叶变换：

$$\mathcal{F}\left(\lambda_l\right)=\hat{f}\left(\lambda_l\right)=\sum_{i=1}^n f(i) u_l(i)=\boldsymbol{u}_l^{\top} \boldsymbol{f}$$

$$\begin{pmatrix} \hat{f}\left(\lambda_1\right) \\ \hat{f}\left(\lambda_2\right) \\ \vdots \\ \hat{f}\left(\lambda_N\right) \end{pmatrix}=\begin{pmatrix} u_1(1) & u_1(2) & \cdots & u_l(N) \\ u_2(1) & u_2(2) & \cdots & u_2(N) \\ \vdots & \vdots & \ddots & \vdots \\ u_N(1) & u_N(2) & \cdots & u_N(N) \end{pmatrix}\begin{pmatrix} f(1) \\ f(2) \\ \vdots \\ f(N) \end{pmatrix} \quad \Longrightarrow \quad \hat{f}=\boldsymbol{U}^{\top} \boldsymbol{f}$$

  - 图傅里叶逆变换
    - $L=U \Lambda U^{\top}$ 为其特征值分解， U 的列向量类比于傅里叶变换中的基 - 图傅里叶逆变换:

$$f(i)=\sum_{l=1}^n \hat{f}\left(\lambda_l\right) u_l(i)=\boldsymbol{u}(i)^{\top} \widehat{\boldsymbol{f}}$$

$$\begin{pmatrix} f(1) \\ f(2) \\ \vdots \\ f(N) \end{pmatrix}=\begin{pmatrix} u_1(1) & u_2(1) & \ldots & u_N(1) \\ u_1(2) & u_1(2) & \ldots & u_N(2) \\ \vdots & \vdots & \ddots & \vdots \\ u_1(N) & u_2(N) & \ldots & u_N(N) \end{pmatrix}\begin{pmatrix} \hat{f}\left(\lambda_1\right) \\ \hat{f}\left(\lambda_2\right) \\ \vdots \\ \hat{f}\left(\lambda_N\right) \end{pmatrix} \quad \Longrightarrow \quad f=\boldsymbol{U} \hat{f}$$

(三）图卷积神经网络

- 谱域上的图卷积
  - 空域上很难定义卷积-->转到谱域
  - 从图信号处理的角度考虑图卷积
  - 卷积公式 $f * g=\mathbb{F}^{-1}\{\mathbb{F}\{f\} \cdot \mathbb{F}\{g\}\}$
  - 卷积定理: 函数卷积的傅里叶 变换是函数傅立叶变换的乘积
  - 给一个图信号 x 和一个卷积核 g

$$x * g=U\left(U^{\top} x \odot U^{\top} g\right)$$

- $U^\top g$ 当成整体的卷积核，用参数 θ 表示

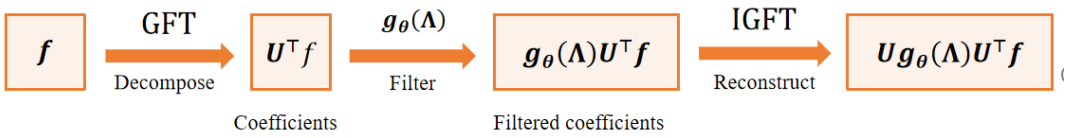
$$x * g = U\left(U^\top x \odot U^\top g\right) = U\left(U^\top x \odot \theta\right) = U g_\theta U^\top x$$

  用 g_θ 表示对角线为 θ 的矩阵

- 谱域图卷积神经网络
  - 回顾

$$\text{GFT: } \hat{\boldsymbol{f}} = \boldsymbol{U}^\top f \quad \text{IGFT: } \boldsymbol{f} = \boldsymbol{U}\hat{\boldsymbol{f}}$$

  - 滤波信号：空域信号 $\rightarrow$ 谱域信号 $\rightarrow$ 滤波 $\rightarrow$ 空域信号



- 最简单的谱域图卷积网络
  - Spectral Graph CNN
    - 无参数化: $\boldsymbol{g}_\theta$ 与 $\boldsymbol{L}$ 的特征向量无关

$$\boldsymbol{x} * \boldsymbol{g} = \boldsymbol{U}\boldsymbol{g}_\theta\boldsymbol{U}^\top\boldsymbol{x} = \boldsymbol{U}\begin{pmatrix}\theta_1 & & & \\ & \theta_2 & & \\ & & \ddots & \\ & & & \theta_n\end{pmatrix}\boldsymbol{U}^\top\boldsymbol{x}$$

  - 缺点
    - 参数多，总共有 n 个参数， n 为节点数
    - 需要对拉普拉斯矩阵进行特征分解， $O\left(n^3\right)$ 时间复杂度
    - 不能局部化：每个节点上信号依赖于其他全部节点信号，而不只是邻居
- 多项式卷积核（ChebyNet ）
  - 局部化：L对图信号f的操作Lf相当于在图上传播一步
  - 多项式卷积核

$$\boldsymbol{g}_\theta(\boldsymbol{\Lambda}) = \sum_{k=0}^{K}\theta_k\boldsymbol{\Lambda}^k$$

$$\boldsymbol{U}\hat{g}(\boldsymbol{\Lambda})\boldsymbol{U}^T\boldsymbol{f} = \boldsymbol{U}\sum_{k=0}^{K}\theta_k\boldsymbol{\Lambda}^k\boldsymbol{U}^T\boldsymbol{f} = \sum_{k=0}^{K}\theta_k\boldsymbol{L}^k\boldsymbol{f}$$

$$\hat{g}(\Lambda) = \begin{bmatrix}\sum_{k=0}^{K}\theta_k\lambda_1^k & & & \\ & \sum_{k=0}^{K}\theta_k\lambda_2^k & & \\ & & \cdots & \\ & & & \sum_{k=0}^{K}\theta_k\lambda_N^k\end{bmatrix}$$

    - 局部性：实际上在图上传播了K步，因此一个节点只影响它周围距离为 K 以内的邻居
    - 效率高：不需要进行特征值分解
    - 参数少：只有 $\mathrm{K}+1$ 个参数
  - 切比雪夫多项式，通过如下递归定义
    - $T_0(x) = 1; T_1(x) = x$
    - $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$
  - 由于其定义域为 $[-1, 1]$ ，因此 $\hat{g}(\Lambda)$ 通过如下方式定义

$$\hat{g}(\boldsymbol{\Lambda}) = \sum_{k=0}^{K}\theta_k T_k(\tilde{\Lambda}) \quad \text{其中 } \tilde{\Lambda} = \frac{2\Lambda}{\lambda_{\max}} - \boldsymbol{I}$$

- 在归一化之后，仍然可以不用计算特征值分解
- 有结论 $\boldsymbol{U}\hat{g}(\boldsymbol{\Lambda})\boldsymbol{U}^{\top}\boldsymbol{f}=\sum_{k=0}^{K}\theta_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{f}$, 其中 $\tilde{\boldsymbol{L}}=\frac{2L}{\lambda_{\max}}-\boldsymbol{I}$
- 归一化图拉普拉斯矩阵 $\boldsymbol{L}^{\mathrm{sym}}$

  最大特征值约为 2，$\lambda_{\max}\approx 2$

$$\boldsymbol{L}^{\mathrm{sym}}=\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{L}\boldsymbol{D}^{-\frac{1}{2}}=\boldsymbol{I}-\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$$

$$\tilde{\boldsymbol{L}}=\frac{2\boldsymbol{L}^{sym}}{\lambda_{\max}}-\boldsymbol{I}=\boldsymbol{L}^{\mathrm{sym}}-\boldsymbol{I}=-\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$$

- 从ChebyNet到GCN
  - 假设我们只取 1 阶切比雪夫多项式，且 $\lambda_{\max}\approx 2$

$$\boldsymbol{y}=\sum_{k=0}^{1}\theta_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{x}\approx\theta_0 T_0(\tilde{\boldsymbol{L}})\boldsymbol{x}+\theta_1 T_1(\tilde{\boldsymbol{L}})\boldsymbol{x}\approx\theta_0\boldsymbol{x}-\theta_1\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{x}$$

$$\text{取 }\theta'=\theta_0=-\theta_1\quad\approx\theta'\left(\boldsymbol{I}+\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}\right)\boldsymbol{x}$$

  - 此时节点只能被它周围的1阶邻接点所影响，但只需要叠加K层这 样的图卷积层，就可以把节点的影响力扩展到K阶邻居节点

    根据类似方法，证明可得 $\boldsymbol{I}+\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$ 的特征值在 $[0,2]$ 之间，叠加多层图卷积层会多次迭代这个操作，可能造成数值不稳定和梯度爆炸的问题

四、实验步骤

1. **数据预处理**

   将字符类型的标签映射为整型类别。设置训练集: 验证集: 测试集 = 0.6: 0.2: 0.2。

2. **定义指标**

   节点分类任务中，cora数据集和citeseer数据集是multi-class，用正确率作为评价指标，ppi数据集为multi-label，用F1指标作为评价指标。

   链路预测任务中，采用AUC作为评价指标。

3. **图网络模型**

```
def _add_self_loops(edge_index, num_nodes):
    """
    添加自环
    """
    loop_index = torch.arange(0, num_nodes, dtype=torch.long,
device=edge_index.device)
    loop_index = loop_index.unsqueeze(0).repeat(2, 1)
    edge_index = torch.cat([edge_index, loop_index], dim=1)
    return edge_index


def _degree(index, num_nodes, dtype):
    """
    计算每个节点的度
    """
    out = torch.zeros((num_nodes), dtype=dtype, device=index.device)
    return out.scatter_add_(0, index, out.new_ones((index.size(0))))
```

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels, add_self_loops=True):
        super(GCNConv, self).__init__(aggr='mean')  # "mean" aggregation
(Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)
        self.add_self_loops = add_self_loops

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        if self.add_self_loops:
            edge_index = _add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        col_deg = _degree(col, x.size(0), dtype=x.dtype)
        row_deg = _degree(row, x.size(0), dtype=x.dtype)
        col_deg_inv_sqrt = col_deg.pow(-0.5)
        row_deg_inv_sqrt = row_deg.pow(-0.5)
        norm = row_deg_inv_sqrt[row] * col_deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)

    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

搭建GCN模型。

函数 `_add_self_loops` 用于在图结构中添加自环。其中，输入参数 `edge_index` 是边的索引矩阵，形状为 `(2, E)`，即每列为一条边的起始点和终止点，共有 E 条边；`num_nodes` 表示节点的数量。

函数 `_degree` 用于计算每个节点的度数。其中，输入参数 `index` 是边的索引矩阵，形状为 `(2, E)`；`num_nodes` 表示节点的数量；`dtype` 表示输出张量的数据类型。

类 `GCNConv` 继承了 PyG 中的 `MessagePassing` 类。它接受节点特征矩阵 `x` 和边的索引矩阵 `edge_index` 作为输入，输出经过 GCN 算法处理后的节点特征矩阵。在 `__init__` 函数中，定义了一个线性变换层 `lin`，输入维度是 `in_channels`，输出维度是 `out_channels`。如果 `add_self_loops` 参数为 True，则调用 `_add_self_loops` 函数添加自环；在 `forward` 函数中，首先判断是否需要添加自环，如果需要，则调用 `_add_self_loops` 函数添加自环。然后通过线性变换层 `lin` 对节点特征进行线性变换。接着计算规范化因子，用于将相邻节点的特征进行归一化。最后调用 `propagate` 函数，开始消息传递过程。在消息传递过程中，通过调用 `message` 函数对每个节点的特征向量进行归一化，再将其乘以规范化因子，最后通过聚合函数 `aggr` 对归一化后的节点特征向量进行聚合，得到新的节点特征矩阵。

`message` 函数用于对每个节点的邻居节点进行消息传递。函数接收一个维度为 [E, out_channels] 的节点特征张量 `x_j` 和一个维度为 [E] 的规范化因子 `norm`，其中 `E` 表示图中的边数。

3. **节点分类**

```
class NodeNet(torch.nn.Module):
    """Network for node classification
    """

    def __init__(self, node_features, hidden_features, num_layers,
num_classes, add_self_loops=True,
                 pair_norm=False, drop_edge=False, act_fn='prelu',dataset =
'cora'):
        super(NodeNet, self).__init__()
        self.pair_norm = pair_norm
        self.drop_edge = drop_edge
        self.convs = nn.ModuleList()
        self.dataset = dataset
        for i in range(num_layers):
            in_channels = hidden_features
            out_channels = hidden_features
            if i == 0:
                in_channels = node_features

            if i == num_layers - 1:
                out_channels = num_classes

            self.convs.append(GCNConv(in_channels=in_channels,
out_channels=out_channels,
                                      add_self_loops=add_self_loops))

        if pair_norm:
            self.pn = PairNorm()

        if act_fn == 'prelu':
            self.act_fn = nn.PReLU()
        elif act_fn == 'relu':
            self.act_fn = nn.ReLU()
        elif act_fn == 'softplus':
            self.act_fn = nn.Softplus()
        else:
            self.act_fn = nn.Tanh()

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        for i, conv in enumerate(self.convs):
            x = conv(x, edge_index)
            if self.pair_norm:
                x = self.pn(x)
            if self.drop_edge:
                edge_index = dropout_edge(edge_index=edge_index, p=0.2)[0]
            if self.dataset == 'ppi':
                x = self.act_fn(x)
            else:
                if i != len(self.convs) - 1:
                    x = self.act_fn(x)
        return x
```

```
class NodeClassification(object):
    def __init__(self, device="cuda", dataset="cora", path="../data/cora/"):
        super(NodeClassification, self).__init__()
        self.net = None
        self.device = torch.device(device)
        self.data = None
        self.path = path
        self.dataset = dataset
        self.loss_list = {"train": [], "val": []}


    def train(self, patience=3, epochs=10, lr=2e-5, hidden_features=16,
num_layers=2, add_self_loops=True,
              pair_norm=False, drop_edge=False, test=False, act_fn='prelu'):
        self.data = load_data(path=self.path, dataset=self.dataset,
task='node')
        num_classes = self.data['num_classes'].item()
        self.data = self.data.to(self.device)
        self.net = NodeNet(node_features=self.data.num_features,
hidden_features=hidden_features,
                           num_classes=num_classes, num_layers=num_layers,
add_self_loops=add_self_loops,
                           pair_norm=pair_norm, drop_edge=drop_edge,
act_fn=act_fn, dataset = self.dataset)
        total_params = sum([param.nelement() for param in
self.net.parameters()])
        print(f">>> total params: {total_params}")
        self.net.to(self.device)
        optimizer = Adam(self.net.parameters(), lr=lr)
        best_model_path = f"./model/{self.dataset}_node_best.pth"
        delay = 0
        best_val_loss = np.inf
        best_val_score = -1
        for epoch in range(epochs):
            self.net.train()
            optimizer.zero_grad()
            out = self.net(self.data)
            train_loss = criterion(out[self.data.train_mask],
self.data.y[self.data.train_mask], self.dataset)
            self.loss_list['train'].append(train_loss.item())
            # train_score = accuracy(out[self.data.train_mask],
self.data.y[self.data.train_mask], self.dataset)
            train_loss.backward()
            optimizer.step()

            with torch.no_grad():
                self.net.eval()
                val_loss = criterion(out[self.data.val_mask],
self.data.y[self.data.val_mask], self.dataset).item()
                self.loss_list['val'].append(val_loss)

                val_score = accuracy(out[self.data.val_mask],
self.data.y[self.data.val_mask], self.dataset)
                # if (epoch % 10) == 0:
                #     print(f"epoch: {epoch}, train_loss:
{train_loss.item():7.5f}, train_score: {train_score:7.5f}, "
```

```
                #           f"val_loss: {val_loss:7.5f}, val_score:
{val_score:7.5f}")

            if val_loss < best_val_loss:
                best_val_loss = val_loss
                best_val_score = val_score
                torch.save(self.net, best_model_path)
                delay = 0
            else:
                delay += 1
                if delay > patience:
                    break


        print(">>> Finished training")
        plot_loss(self.loss_list, 'Node Classification', self.dataset)
        print(">>> Finished plot loss")
        print(f"best_val_loss: {best_val_loss:7.4f}, best_val_score:
{best_val_score:7.4f} \n"
              f"{best_val_loss:7.4f} | {best_val_score:7.4f} |")

        if test:  # whether test on test dataset
            self.net = torch.load(best_model_path)
            self.net.to(self.device)
            with torch.no_grad():
                self.net.eval()
                out = self.net(self.data)
                test_loss = criterion(out[self.data.test_mask],
self.data.y[self.data.test_mask], self.dataset).item()
                test_score = accuracy(out[self.data.test_mask],
self.data.y[self.data.test_mask], self.dataset)
                print(f"test_score {test_score:7.4f}")
```

用搭建好的 GCN 模型进行节点分类。

定义了`NodeNet`类，实现了用于节点分类的 GCN 模型，它含有多个 GCNConv 层。其中，每一GCNConv 层都会计算节点之间的信息传递并更新节点的特征表征，进而实现对节点的分类。在前向传播过程中，模型将输入数据x和网络拓扑结构 edge_index 带入每个 GCNConv 层中进行更新。其中，如果需要进行归一化，则使用`PairNorm`函数进行处理；如果需要丢弃部分边信息，则使用`dropout_edge`函数随机删除一部分边。最后，如果不是最后一层，则对输出结果进行激活函数处理调整维度并进入下一层计算，否则直接返回输出结果。

定义了`NodeClassification`类。在 train 方法中，首先使用 load_data 函数加载数据集，然后根据输入的参数创建了一个 NodeNet 对象，该对象包含了基于 GCN 的节点分类网络模型，并将模型装载至指定 device 上。接着定义了损失函数、优化器以及最佳模型保存位置等参数，并进入训练循环。在每个 epoch 中，首先对训练数据进行前向传播并计算损失函数，然后进行反向传播以及模型参数更新，接着对验证集进行评估并记录相关信息。在连续超过 patience 轮未出现新的最佳模型时停止训练，同时输出最佳验证集损失以及准确率等结果，并通过可视化展示训练过程中的损失函数。最后，如果 test 参数为 True，则在测试集上对训练后的模型进行预测并计算相应的准确率。

4. **链路预测**

```
class LinkNet(torch.nn.Module):
    """Network for link prediction
    """
```

```
    def __init__(self, node_features, hidden_features, num_layers,
add_self_loops=True, pair_norm=False,
                 drop_edge=False, act_fn='prelu') -> None:
        super(LinkNet, self).__init__()
        self.pair_norm = pair_norm
        self.drop_edge = drop_edge
        self.convs = nn.ModuleList()
        for i in range(num_layers):
            if i == 0:
                self.convs.append(
                    GCNConv(in_channels=node_features,
out_channels=hidden_features, add_self_loops=add_self_loops))
            else:
                self.convs.append(
                    GCNConv(in_channels=hidden_features,
out_channels=hidden_features, add_self_loops=add_self_loops))
        if self.pair_norm:
            self.pn = PairNorm()

        if act_fn == 'prelu':
            self.act_fn = nn.PReLU()
        elif act_fn == 'relu':
            self.act_fn = nn.ReLU()
        elif act_fn == 'softplus':
            self.act_fn = nn.Softplus()
        else:
            self.act_fn = nn.Tanh()

    def encode(self, x, edge_index):
        for i, conv in enumerate(self.convs):
            x = conv(x, edge_index)
            if self.pair_norm:
                x = self.pn(x)
            if self.drop_edge:
                edge_index = dropout_edge(edge_index=edge_index, p=0.2)[0]
            if i != len(self.convs) - 1:
                x = self.act_fn(x)
        return x

    def decode(self, z, edge_index):
        # edge_index = torch.cat([pos_edge_index, neg_edge_index], dim=-1)
#[2, E]
        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()


class LinkPrediction(object):
    def __init__(self, device="cuda", dataset="cora", path="../data/cora/")
-> None:
        super(LinkPrediction, self).__init__()
        self.net = None
        self.device = torch.device(device)
        self.data = None
        self.path = path
```

```
        self.dataset = dataset
        self.loss_list = {"train": [], "val": []}


    def get_link_labels(self, pos_edge_index, neg_edge_index):
        num_links = pos_edge_index.size(1) + neg_edge_index.size(1)
        link_labels = torch.zeros(num_links, dtype=torch.float)
        link_labels[:pos_edge_index.size(1)] = 1
        return link_labels

    def train(self, patience=3, epochs=10, lr=2e-5, hidden_features=16,
num_layers=2, add_self_loops=True,
              pair_norm=False, drop_edge=False, test=False, act_fn='prelu'):
        self.data = load_data(self.path, self.dataset, 'link')
        self.data = self.data.to(self.device)
        self.net = LinkNet(self.data.num_features, hidden_features,
num_layers, add_self_loops, pair_norm, drop_edge,
                           act_fn=act_fn)
        total_params = sum([param.nelement() for param in
self.net.parameters()])
        print(f">>> total params: {total_params}")
        self.net.to(self.device)
        optimizer = Adam(self.net.parameters(), lr=lr)
        best_model_path = f"./model/{self.dataset}_link_best.pth"
        delay = 0
        best_val_loss = np.inf
        best_val_score = -1
        for epoch in range(epochs):
            neg_edge_index =
negative_sampling(edge_index=self.data.train_pos_edge_index,

num_nodes=self.data.num_nodes,

num_neg_samples=self.data.train_pos_edge_index.size(1))
            self.net.train()
            optimizer.zero_grad()
            z = self.net.encode(self.data.x, self.data.train_pos_edge_index)
            edge_index = torch.cat([self.data.train_pos_edge_index,
neg_edge_index], dim=-1)
            link_logits = self.net.decode(z, edge_index)
            link_labels =
self.get_link_labels(self.data.train_pos_edge_index,
neg_edge_index).to(self.data.x.device)

            train_loss = criterion(link_logits, link_labels,
dataset=self.data, task='link')
            # train_score = accuracy(link_logits.sigmoid(), link_labels,
self.dataset, 'link')
            self.loss_list['train'].append(train_loss.item())

            train_loss.backward()
            optimizer.step()

            with torch.no_grad():
                self.net.eval()

                z = self.net.encode(self.data.x,
self.data.train_pos_edge_index)
```

```
                edge_index = self.data.val_edge_index

                link_logits = self.net.decode(z, edge_index)
                link_probs = link_logits.sigmoid()
                link_labels = self.data.val_edge_label
                val_loss = criterion(link_logits, link_labels, self.dataset,
'link').item()
                self.loss_list['val'].append(val_loss)
                val_score = accuracy(link_probs, link_labels, self.dataset,
'link')
                # if (epoch % 10) == 0:
                    # print(f"epoch: {epoch}, train_loss:
{train_loss.item():7.5f}, train_score: {train_score:7.5f}, "
                    #       f"val_loss: {val_loss:7.5f}, val_score:
{val_score:7.5f}")

                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    best_val_score = val_score
                    torch.save(self.net, best_model_path)
                    delay = 0
                else:
                    delay += 1
                    if delay > patience:
                        break


        print(">>> Finished training")
        plot_loss(self.loss_list, 'Link Prediction', self.dataset)
        print(">>> Finished plot loss")
        print(f"best_val_loss: {best_val_loss:7.4f}, best_val_score:
{best_val_score:7.4f} \n"
              f"{best_val_loss:7.4f} | {best_val_score:7.4f} |")
        if test:  # whether test on test dataset
            self.net = torch.load(best_model_path)
            self.net.to(self.device)
            with torch.no_grad():
                self.net.eval()
                z = self.net.encode(self.data.x,
self.data.train_pos_edge_index)
                edge_index = self.data.test_edge_index

                link_logits = self.net.decode(z, edge_index)
                link_probs = link_logits.sigmoid()
                link_labels = self.data.test_edge_label
                test_loss = criterion(link_logits, link_labels,
self.dataset, 'link')
                test_score = accuracy(link_probs, link_labels, self.dataset,
'link')
                print(f"test_score {test_score:7.4f}")
```

用搭建好的GCN 模型进行链路预测。

定义`LinkNet`类。实现学习节点嵌入并进行链接预测的GCN模型。该模型包含了 encode 和 decode 两个方法。其中，encode 方法对节点嵌入进行学习，通过多层 GCNConv 来对节点特征进行聚合，其中第一层的输入通道数为 node_features，输出通道数为 hidden_features，剩余的层数的输入和输出通道数均为 hidden_features。如果启用了 pair_norm，则对节点嵌入进行规范

化，如果启用了 drop_edge，则对边进行随机 dropout。最后一层的输出被作为节点的嵌入表示返回；decode 方法根据节点嵌入和边列表来预测边的存在性。输入参数 z 是节点嵌入的表示，edge_index 是边的索引。该方法会返回一个大小为 edge_index 的张量，表示哪些边存在；decode_all 方法接受节点嵌入的表示 z，并返回一个大小为 [N, N] 的张量，其中 N 是图中节点的数量。该张量的 (i,j) 元素的值表示从节点 i 到节点 j 是否存在链接，即 1 表示存在链接，0 表示不存在链接。

定义`LinkPrediction`类，使用 LinkNet 模型。train 过程中，首先使用正样本边和负样本边进行采样，然后将这些样本输入到神经网络中。接着计算损失函数，并使用反向传播和优化器进行参数更新。训练结束后，如果需要，可以在测试数据集上进行评估。在最终测试评估期间输出了测试分数，用于评估Link Prediction模型的性能。

5. **测试性能**

选择在验证集上表现最好的一组超参数，重新训练模型，在测试集上测试，并记录测试的结果。

四、实验过程与分析

(一) 节点分类

- **Cora**

add_self_loops	epoch	val acc
False	500	0.7269
False	1000	0.7066
True	500	0.8137
True	1000	0.8506

实验可知，在 Cora 数据集上添加自环会提升模型的效果，且设置 epoch=1000 较合适。

num_layers	val acc
1	0.8579
2	0.8506
3	0.8210

实验可知，在 Cora 数据集上一层 GCN 的效果已经足够。

DropEdge	val acc
False	0.8579
True	0.8339

实验可知，在 Cora 数据集上DropEdge不会提升模型的效果。

pair_norm	val acc
False	0.8579
True	0.7915

实验可知，在 Cora 数据集上pair_norm不会提升模型的效果。

act_fn	val acc
relu	0.8561
prelu	0.8579
softplus	0.8469
tanh	0.8229

实验可知，在 Cora 数据集上激活函数使用 prelu 效果较好。

- Citeseer

add_self_loops	epochs	val acc
False	500	0.5227
True	500	0.7628

实验可知，在 Citeseer 数据集上添加自环会提升模型的效果。

num_layers	epoch	val acc
1	500	0.7628
1	1000	0.7598
2	500	0.7432
2	1000	0.7523

实验可知，在 Citeseer 数据集上1层 GCN ， epoch = 500 效果较好。

DropEdge	val acc
False	0.7628
True	0.7538

实验可知，在 Citeseer 数据集上DropEdge不会提升模型的效果。

pair_norm	val acc
False	0.7628
True	0.7221

实验可知，在 Citeseer 数据集上pair_norm不会提升模型的效果。

act_fn	val acc
relu	0.7583
prelu	0.7628
softplus	0.7356
tanh	0.7644

实验可知，在 Citeseer 数据集上几种激活函数的效果差别不大，tanh 较好。

- PPI

add_self_loops	epochs	val acc
False	500	0.3368
True	500	0.3572

实验可知，在 PPI 数据集上添加自环会提升模型的效果。

DropEdge	val acc
False	0.3572
True	0.3574

实验可知，在 PPI 数据集上DropEdge与否增效不显著。

pair_norm	val acc
False	0.3574
True	0.4794

实验可知，在 PPI 数据集上pair_norm会提升模型的效果。

act_fn	val acc
prelu	0.4794
tanh	0.4780

实验可知，在 PPI 数据集上两种激活函数的效果相差不大。

num_layers	lr	hidden_features	epoch	val acc
1	1e-3	64	500	0.4794
1	1e-2	64	500	0.4780
1	1e-3	64	1000	0.4774
1	1e-3	128	500	0.4776
2	1e-3	64	500	0.4634
2	1e-3	64	1000	0.4650
2	1e-3	128	500	0.4755
2	1e-2	128	500	0.4866
2	1e-1	128	500	0.4943
3	1e-2	128	500	0.4732

实验可知，在 PPI 数据集上，选择num_layers=2，lr=1e-1，hidden_features=128，epoch=500 的效果较好。

（二）链路预测

- **Cora**

add_self_loops	epoch	val acc
False	500	0.6882
True	500	0.8972

实验时出现了早停，故不再增加epoch。

实验可知，在 Cora 数据集上添加自环会提升模型的效果。

num_layers	val acc
1	0.8972
2	0.5408
3	0.5110

实验可知，在 Cora 数据集上一层 GCN 的效果已经足够。

DropEdge	val acc
False	0.8972
True	0.8990

实验可知，在 Cora 数据集上DropEdge与否增效不明显。

pair_norm	val acc
False	0.8990
True	0.7143

实验可知，在 Cora 数据集上pair_norm不会提升模型的效果。

act_fn	val acc
relu	0.8956
prelu	0.8990
softplus	0.8959
tanh	0.8952

实验可知，在 Cora 数据集上激活函数使用 prelu 效果较好。

- Citeseer

add_self_loops	epochs	val acc
False	500	0.6531
True	500	0.9362

实验时出现了早停，故不再增加epoch。

实验可知，在 Citeseer 数据集上添加自环会提升模型的效果。

num_layers	val acc
1	0.9362
2	0.7847
3	0.5462

实验可知，在 Citeseer 数据集上一层 GCN 的效果已经足够。

DropEdge	val acc
False	0.9362
True	0.9238

实验可知，在 Citeseer 数据集上DropEdge不会提升模型的效果。

pair_norm	val acc
False	0.9362
True	0.8215

实验可知，在 Citeseer 数据集上pair_norm不会提升模型的效果。

act_fn	val acc
relu	0.9346
prelu	0.9362
softplus	0.9343
tanh	0.9358

实验可知，在 Citeseer 数据集上几种激活函数的效果相差不大。

- PPI

add_self_loops	epochs	val acc
False	500	0.5418
True	500	0.5446

实验可知，在 PPI 数据集上添加自环与否增效不显著。

num_layers	val acc
1	0.5446
2	0.3620
3	0.3353

实验可知，在 PPI 数据集上一层 GCN 的效果已经足够。

DropEdge	val acc
False	0.5446
True	0.5470

实验可知，在 PPI 数据集上DropEdge不会提升模型的效果。

pair_norm	val acc
False	0.5446
True	0.7571

实验可知，在 PPI 数据集上pair_norm会显著提升模型的效果。

act_fn	val acc
relu	0.7579
prelu	0.7581
softplus	0.7576
tanh	0.7582

实验可知，在 PPI 数据集上几种激活函数的效果相差不大。

由以上实验过程可知，对每个任务和数据集，自环都会提升模型的效果。因为节点的聚合表征不包含它自己的特征。该表征是相邻节点的特征聚合，因此只有具有自环的节点才会在该聚合中包含自己的特征；层数一般取1-2层效果较好；DropEdge 、PairNorm 和激活函数等在不同的数据集或不同的任务上有不同的表现。

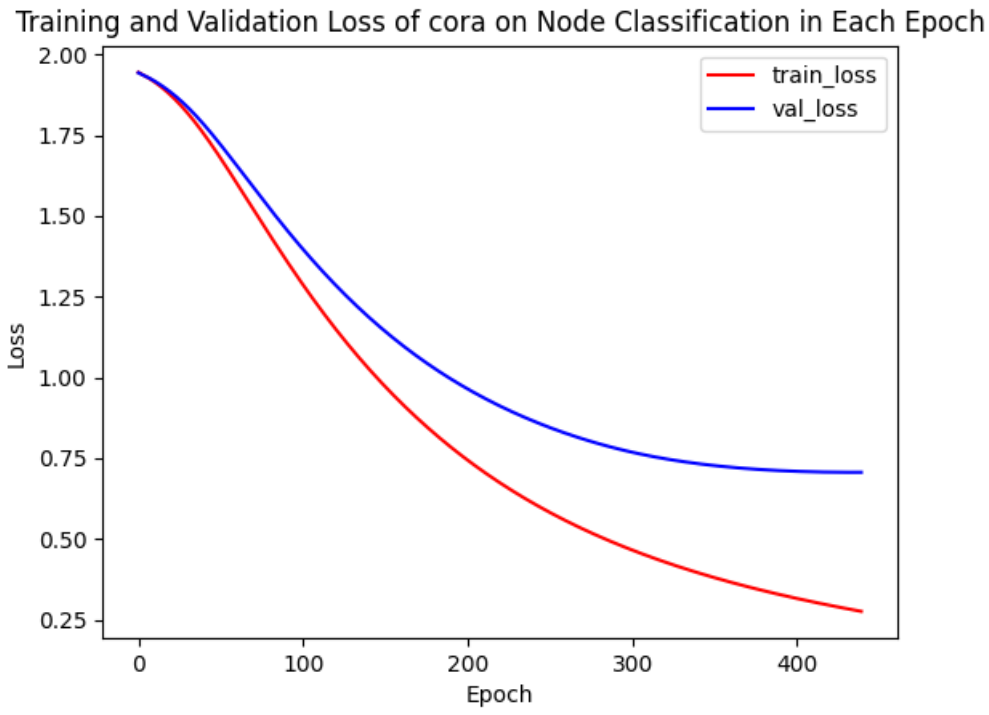
四、实验结果

（一）节点分类

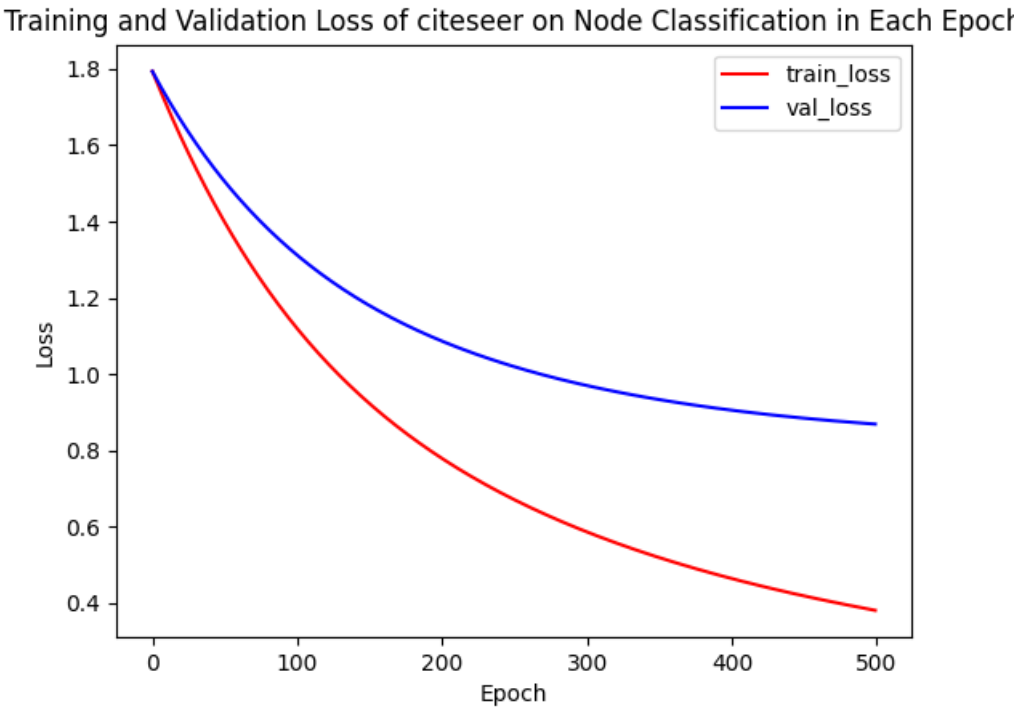
- 最合适的一组超参数&模型在测试集上的结果

数据集	add_self_loops	num_layers	drop_edge	pair_norm	act_fn	test score
Cora	True	1	False	False	prelu	0.8635
Citeseer	True	1	False	False	tanh	0.7541
PPI	True	2	True	True	prelu	0.5130

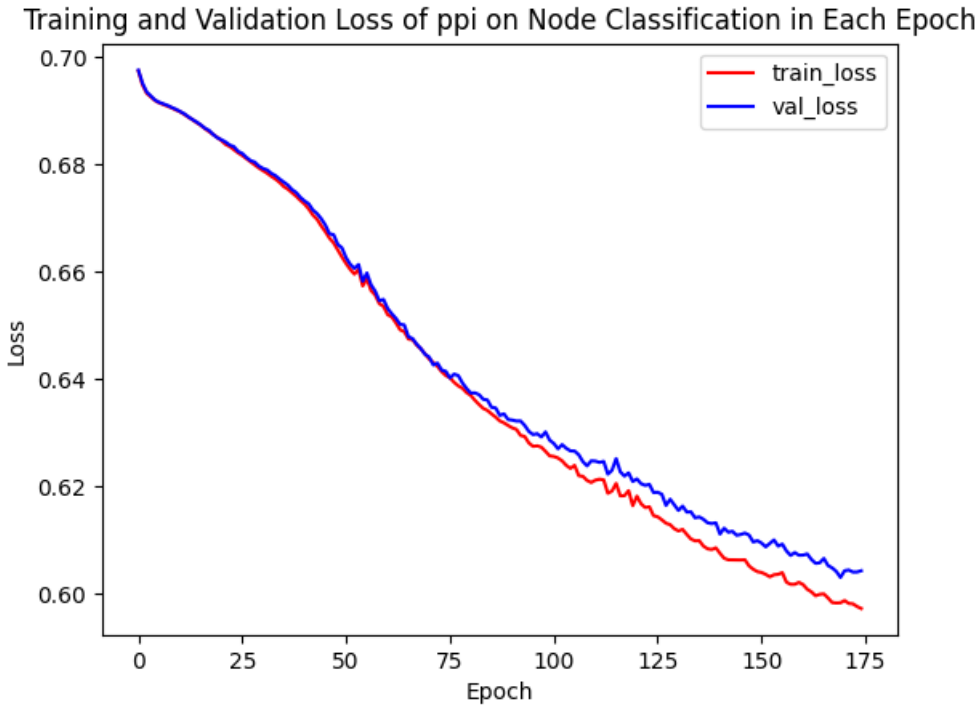
- 模型在训练集和验证集上的损失及可视化图
  - Cora



- Citeseer



- PPI

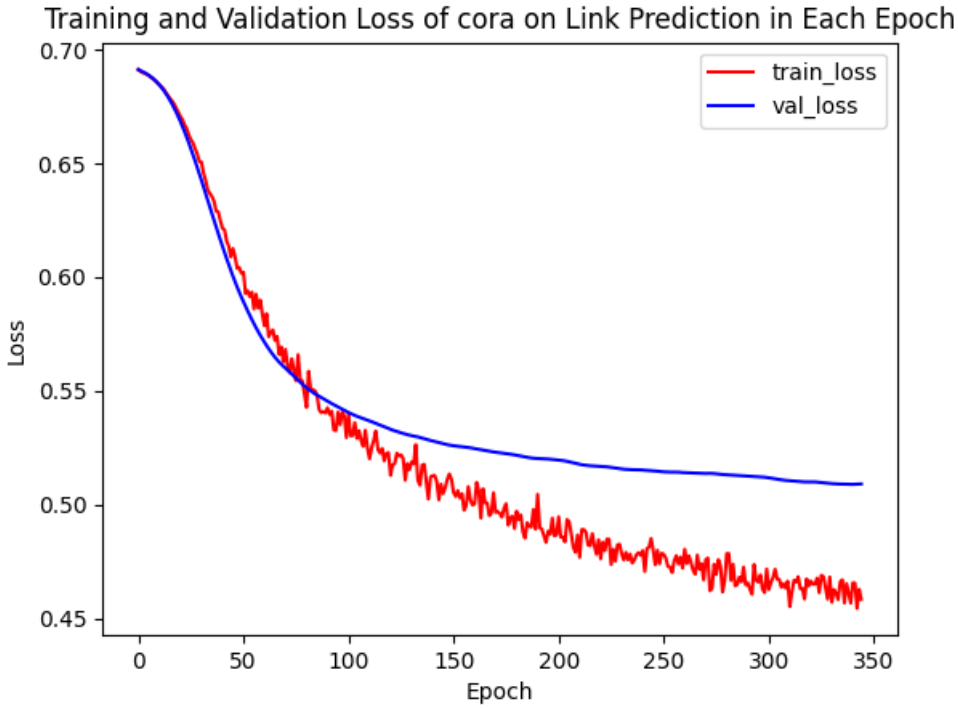


（二）链路预测

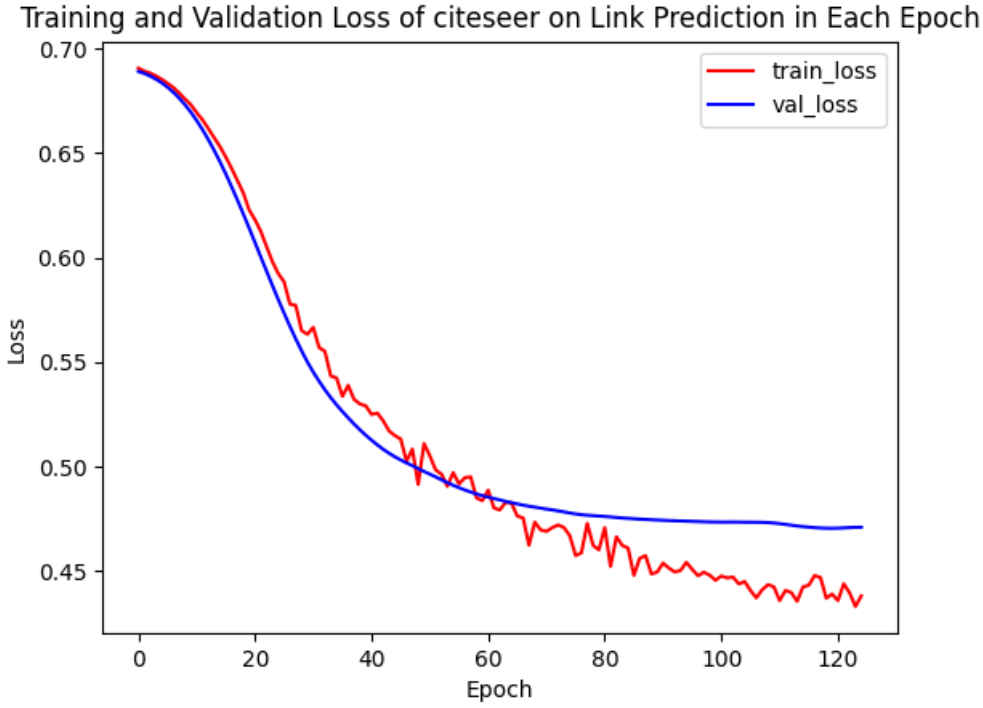
- 最合适的一组超参数&模型在测试集上的结果

数据集	add_self_loops	num_layers	drop_edge	pair_norm	act_fn	test score
Cora	True	1	True	False	prelu	0.8990
Citeseer	True	1	False	False	prelu	0.9353
PPI	True	1	False	True	tanh	0.7586

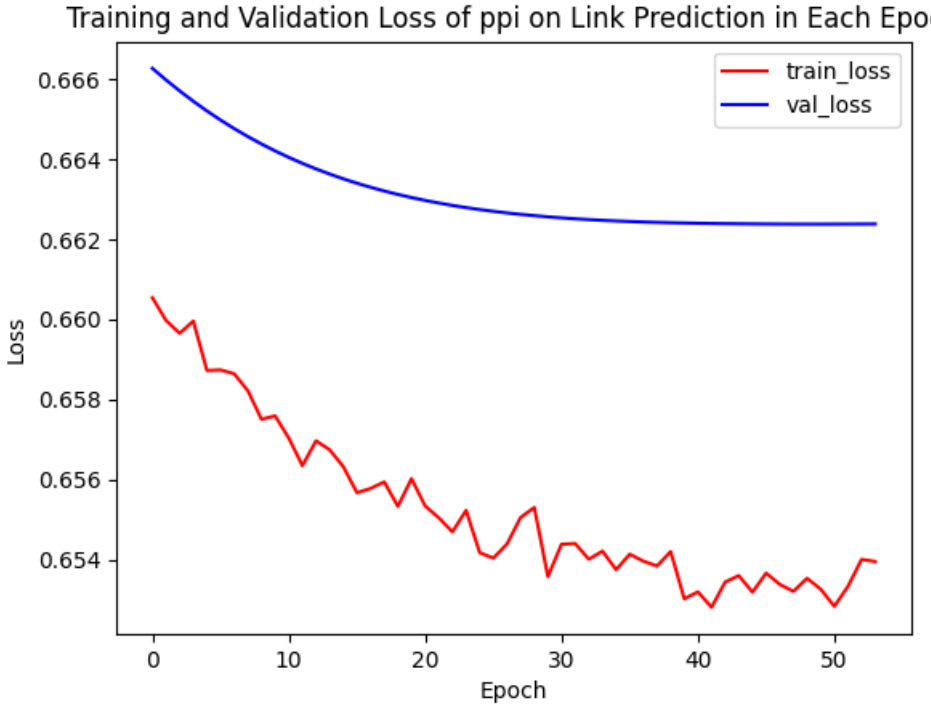
- 模型在训练集和验证集上的损失及可视化图
  - Cora



  - Citeseer



  - PPI


Training and Validation Loss of ppi on Link Prediction in Each Epoch
train_loss
val_loss
0.666
0.664
0.662
0.660
0.658
0.656
0.654
Loss
0
10
20
30
40
50
Epoch