GCN实现节点分类与链路预测任务

陆叶青 PB21081591

实验目的

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 。

实验过程

1. 网络搭建

本实验环境配置PyTorch 2.0.0, Python 3.8(ubuntu20.04), Cuda 11.8 。

2. 数据集以及数据预处理

本次实验使用的数据包含两个常用的图结构数据集：Cora 、Citeseer 。

Cora数据集的参数如下：

```
NumNodes: 2708
NumEdges: 10556
NumFeats: 1433
NumClasses: 7
NumTrainingSamples: 140
NumValidationSamples: 500
NumTestSamples: 1000
```

Citeseer 数据集的参数如下：

```
NumNodes: 3327
NumEdges: 9228
NumFeats: 3703
NumClasses: 6
NumTrainingSamples: 120
NumValidationSamples: 500
NumTestSamples: 1000
```

笔者调用dgl库进行数据下载，通过下面的Load_graph得到表示相应数据集的图。

```
def Load_graph(data : str):
    if data == 'cora':
        dataset = dgl.data.CoraGraphDataset(raw_dir="./Datasets/DGL/")

    elif data == 'citeseer':
        dataset = dgl.data.CiteseerGraphDataset(raw_dir="./Datasets/DGL/")
    else:
        raise Exception('unknown data')
    return dataset

dataset = Load_graph("cora")
g = dataset[0]
```

3. 模型搭建

根据GCN前向传播的公式，笔者通过自定义的GCNLayer类实现GCN层的网络架构。

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

其中，$\tilde{A} = A + I$，I是单位矩阵,A是图的邻接矩阵，$\tilde{D}$是$\tilde{A}$的度矩阵，H是每一层的特征，σ是非线性激活函数。

```
class GCNLayer(nn.Module):
    def __init__(self,in_feats,out_feats,bias=True):
        super(GCNLayer,self).__init__()
        self.weight = nn.Parameter(torch.Tensor(in_feats,out_feats))
        if bias:
            self.bias = nn.Parameter(torch.zeros(out_feats))
        else:
            self.bias = None

        self.reset_parameter()

    def reset_parameter(self):#初始化weight矩阵
        nn.init.xavier_uniform_(self.weight)

    def forward(self,g,h):
        with g.local_scope():
            # H * W
            h = torch.matmul(h,self.weight)
            # D^{-1/2} * H * W
            g.ndata['h'] = h * g.ndata['norm']
            #对应的是AH的计算，即将每个节点的特征通过邻接关系传播给邻居节点，相当于邻接矩阵与节点特征矩阵相乘
            # A * D^{-1/2} * H * W
            g.update_all(message_func = fn.copy_u('h','m'),
                         reduce_func=fn.sum('m','h'))
            h = g.ndata['h']
            # D^{-1/2} * A * D^{-1/2} * H * W
            h = h * g.ndata['norm']
            if self.bias is not None:
                h = h + self.bias
            return h
```

```
#求度矩阵
degs = g.out_degrees().float()
#计算 D^{-1/2}
norm = torch.pow(degs, -0.5)
norm[torch.isinf(norm)] = 0
g.ndata['norm'] = norm.unsqueeze(1)
```

然后为了笔者定义了链路预测和节点分类的神经网络GCNModel。

```
class PairNorm(nn.Module):
    def __init__(self, scale=1):
        super(PairNorm, self).__init__()
        self.scale = scale

    def forward(self, x):
        col_mean = x.mean(dim=0)
        rownorm_individual = (1e-6 + x.pow(2).sum(dim=1, keepdim=True)).sqrt()#
范数
        x = self.scale * x / rownorm_individual - col_mean# 归一化后减去均值
        return x

class GCNModel(nn.Module):
    def __init__(self, in_feats, h_feats, num_classes, num_layers, bias,
pair_norm, activation, dropedge=False, p=0):
        super(GCNModel_for_NC,self).__init__()
        self.bias = bias
        self.pair_norm = pair_norm
        self.num_layers = num_layers
        self.activation = activation
        self.dropedge = dropedge
        self.p = p
        self.conv1 = GCNLayer(in_feats,h_feats,bias)
        self.conv2 = GCNLayer(h_feats,h_feats,bias)
        self.conv3 = GCNLayer(h_feats,num_classes,bias)

    def forward(self, g, in_feat):
        if self.dropedge:
            g = dgl.transforms.DropEdge(p=self.p)(g)
        for i in range(self.num_layers):
            if i == 0:
                h = self.conv1(g, in_feat)
                if self.pair_norm:
                    h = PairNorm()(h)
                h = self.activation(h)
            elif i == self.num_layers - 1:
                h = self.conv3(g, h)
            else:
                h = self.conv2(g, h)
                if self.pair_norm:
                    h = PairNorm()(h)
                h = self.activation(h)
        return h
```

4. 模型训练

（1）节点分类

在训练节点分类任务的模型时，笔者使用如下的优化方法和损失函数。在训练过程中，记录训练集和验证集的损失和准确率，用于做出训练损失函数图像。

```
def train(g, model, train_mask, val_mask, num_epochs, lr, wd):
    model.train()
    best_val_acc = 0
    features, labels = g.ndata['feat'], g.ndata['label']
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=wd)
    train_acc_list, val_acc_list = [], []
    train_loss_list = []
    val_loss_list = []
    for epoch in range(num_epochs):
        logits = model(g, features)  # (N, label_nums)
        pred = logits.argmax(1)
        train_loss = criterion(logits[train_mask], labels[train_mask])
        val_loss = criterion(logits[val_mask], labels[val_mask])
        train_acc = (pred[train_mask] == labels[train_mask]).float().mean()
        val_acc = (pred[val_mask] == labels[val_mask]).float().mean()
        train_acc_list.append(train_acc.item())
        val_acc_list.append(val_acc.item())
        train_loss_list.append(train_loss.item())
        val_loss_list.append(val_loss.item())
        if best_val_acc < val_acc:
            best_val_acc = val_acc

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if (epoch + 1) % 5 == 0:
            print(
                'Epoch {}, train loss: {:.3f}, train acc: {:.3f}, val acc: {:.3f}
(best {:.3f}) '
                .format(epoch + 1, train_loss, train_acc, val_acc, best_val_acc
                        ))
    return train_loss_list, train_acc_list, val_loss_list, val_acc_list

def test(g, model, test_mask):
    model.eval()
    features, labels = g.ndata['feat'], g.ndata['label']
    criterion = nn.CrossEntropyLoss()
    with torch.no_grad():
        logits = model(g, features)  # (N, label_nums)
        pred = logits.argmax(1)
        test_loss = criterion(logits[test_mask], labels[test_mask])
        test_acc = (pred[test_mask] == labels[test_mask]).float().mean()
    return test_loss.item(), test_acc.item()
```

（2）链路预测

在训练链路预测任务的模型时，笔者首先对图的边集进行划分。对图中所有真实存在的边，即正边，按照8：1：1的比例划分为训练集、验证集和测试集。然后再得到图中所有不存在的边，即负边，同样按照8：1：1的比例划分为训练集、验证集和测试集。最后将这些边重新组织成图，最终得到的训练集、验证集和测试集里面由正负比例相等的正样本和负样本，从而可以把链路预测问题看成是二分类问题。

```
def edge_partition(g, sloop=False):
    ## 考虑加入自环
    if sloop:
        g = dgl.add_self_loop(g)
    # 分测试集和训练集
    u, v = g.edges()
    eids = np.arange(g.number_of_edges())
    eids = np.random.permutation(eids)
    test_size = int(len(eids) * 0.1)
    val_size = int(len(eids) * 0.1)
    train_size = g.number_of_edges() - test_size - val_size
    test_pos_u, test_pos_v = u[eids[:test_size]], v[eids[:test_size]]
    val_pos_u, val_pos_v = u[eids[test_size:test_size + val_size]],
v[eids[test_size:test_size + val_size]]
    train_pos_u, train_pos_v = u[eids[test_size + val_size:]], v[eids[test_size +
val_size:]]
    # 负边
    adj = sp.coo_matrix((np.ones(len(u)), (u.numpy(), v.numpy())))
    adj_neg = 1 - adj.todense() - np.eye(g.number_of_nodes())
    neg_u, neg_v = np.where(adj_neg != 0)

    neg_eids = np.random.choice(len(neg_u), g.number_of_edges())
    test_neg_u, test_neg_v = neg_u[neg_eids[:test_size]],
neg_v[neg_eids[:test_size]]
    val_neg_u, val_neg_v = neg_u[eids[test_size:test_size +
val_size]],neg_v[eids[test_size:test_size + val_size]]
    train_neg_u, train_neg_v = neg_u[neg_eids[test_size + val_size:]],
neg_v[neg_eids[test_size + val_size:]]

    train_g = dgl.remove_edges(g, eids[:test_size + val_size])

    train_pos_g = dgl.graph((train_pos_u, train_pos_v),
num_nodes=g.number_of_nodes())
    train_neg_g = dgl.graph((train_neg_u, train_neg_v),
num_nodes=g.number_of_nodes())

    val_pos_g = dgl.graph((val_pos_u, val_pos_v), num_nodes=g.number_of_nodes())
    val_neg_g = dgl.graph((val_neg_u, val_neg_v), num_nodes=g.number_of_nodes())

    test_pos_g = dgl.graph((test_pos_u, test_pos_v),
num_nodes=g.number_of_nodes())
    test_neg_g = dgl.graph((test_neg_u, test_neg_v),
num_nodes=g.number_of_nodes())

    return train_g, train_pos_g, train_neg_g, val_pos_g, val_neg_g, test_pos_g,
test_neg_g
```

接着笔者根据链路预测的启发式算法（基于打分的方法），前提假设就是更“相似”的节点更容易产生链接。笔者使用的打分方法是两个点之间的点积，来预测两个节点表示之间的得分，从而来判断两个节点之间存在边的可能性。

```
class DotPredictor(nn.Module):
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return g.edata['score'][:, 0]
```

笔者基于此二分类问题，自定义了损失和auc的计算函数。

```
def compute_loss(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score])
    labels = torch.cat([torch.ones(pos_score.shape[0]),
torch.zeros(neg_score.shape[0])])
    return F.binary_cross_entropy_with_logits(scores, labels)

def compute_auc(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score]).numpy()
    labels = torch.cat(
        [torch.ones(pos_score.shape[0]),
torch.zeros(neg_score.shape[0])]).numpy()
    return roc_auc_score(labels, scores)
```

最后定义训练与测试函数。

```
def train(model, train_g, train_pos_g, train_neg_g, val_pos_g, val_neg_g):
    pred = DotPredictor()
    optimizer = torch.optim.Adam(itertools.chain(model.parameters(),
pred.parameters()), lr=0.01)
    model.train()
    train_loss_list = []
    val_loss_list = []
    val_acc_history = []
    train_acc_history = []
    for e in range(100):
        # forward
        h = model(train_g, train_g.ndata['feat'])
        pos_score = pred(train_pos_g, h)
        neg_score = pred(train_neg_g, h)
        val_pos_score = pred(val_pos_g, h)
        val_neg_score = pred(val_neg_g, h)
        loss = compute_loss(pos_score, neg_score)
        val_loss = compute_loss(val_pos_score, val_neg_score)
        train_auc = test (model, train_pos_g, train_neg_g, h)
        val_auc = test (model, val_pos_g, val_neg_g, h)
        train_loss_list.append(loss.item())
        val_loss_list.append(val_loss.item())
        val_acc_history.append(val_auc)
        train_acc_history.append(train_auc)
```

```
        # backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if e % 5 == 0:
            print("Epoch {:03d}: train_Loss {:.4f},train_Auc {:.4f}, val_Loss
{:.4f},ValAcc {:.4f}".format(
                e, loss.item(), train_auc, val_loss.item(), val_auc))
    return train_loss_list, train_acc_history, val_loss_list, val_acc_history

def test (model, pos_g, neg_g, h):
    model.eval()
    pred = DotPredictor()
    with torch.no_grad():
        pos_score = pred(pos_g, h)
        neg_score = pred(neg_g, h)
        auc = compute_auc(pos_score, neg_score)
    return auc
```

实验结果

1. 节点分类

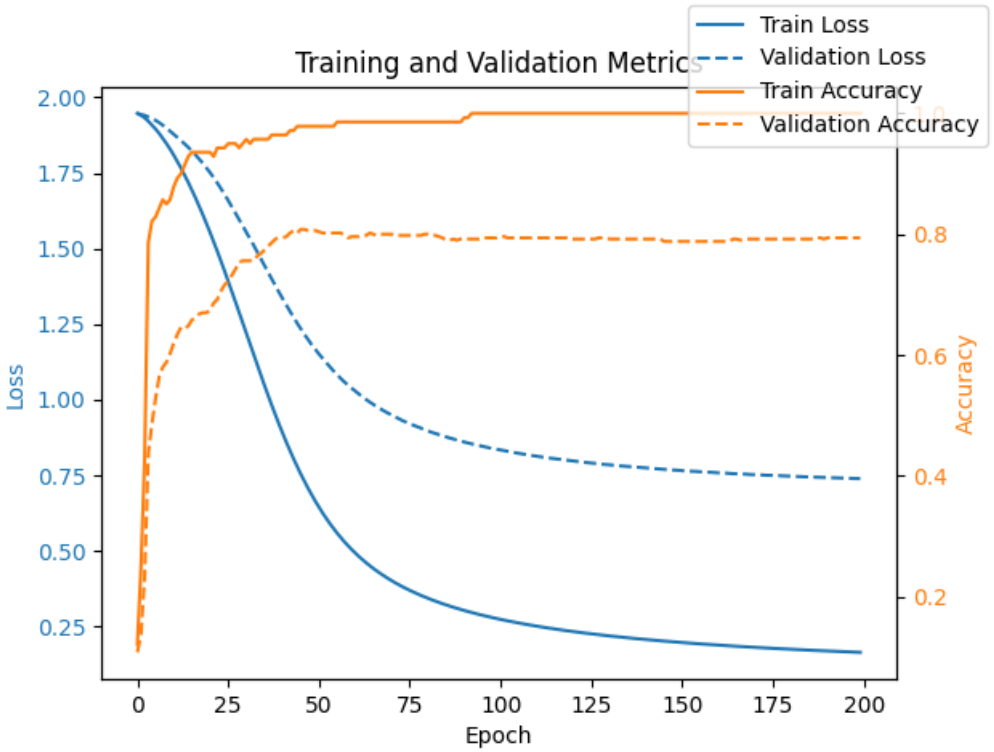
（1）cora数据集

笔者使用两层GCNLayer，激活函数为relu函数，使用自环，不使用dropedge以及pairnorm， epochs = 200，hidden_size = 32， lr = 0.01， wd = 5e-4.

结果如下：

```
loss on test set:0.7094777226448059
accuracy on test set:0.8080000281333923
```

训练过程图像如下：



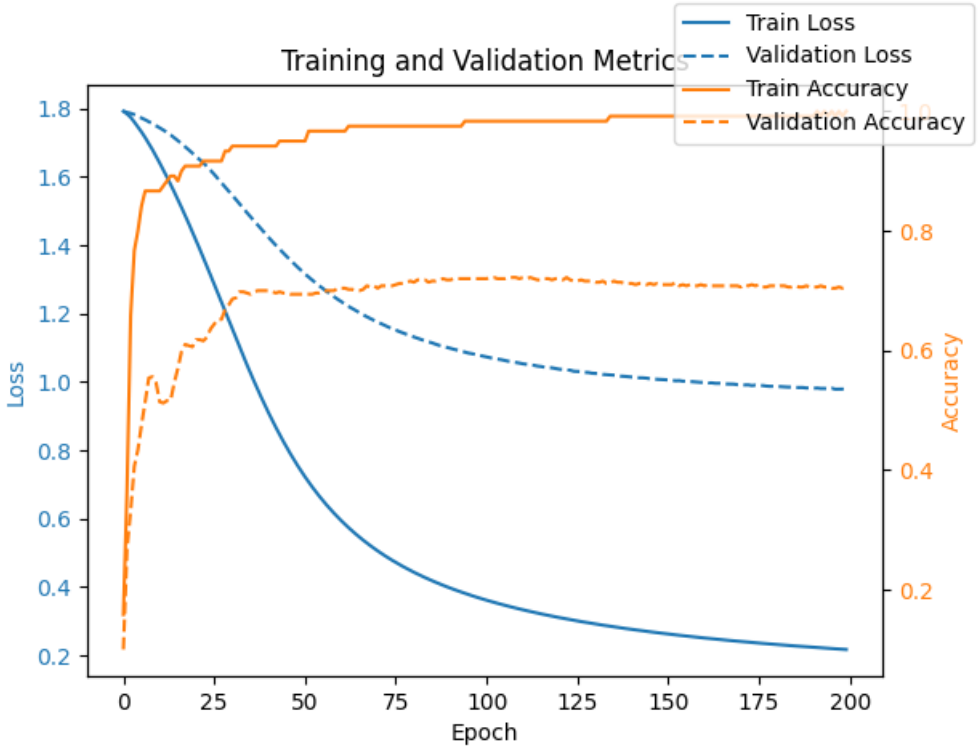
（2）citeseer数据集

笔者使用两层GCNLayer，激活函数为relu函数，使用自环，不使用dropedge以及pairnorm，epochs = 200，hidden_size = 32， lr = 0.01， wd = 5e-4.

结果如下：

```
loss on test set:0.9551671743392944
accuracy on test set:0.7179999947547913
```

训练过程图像如下：



2. 链路预测

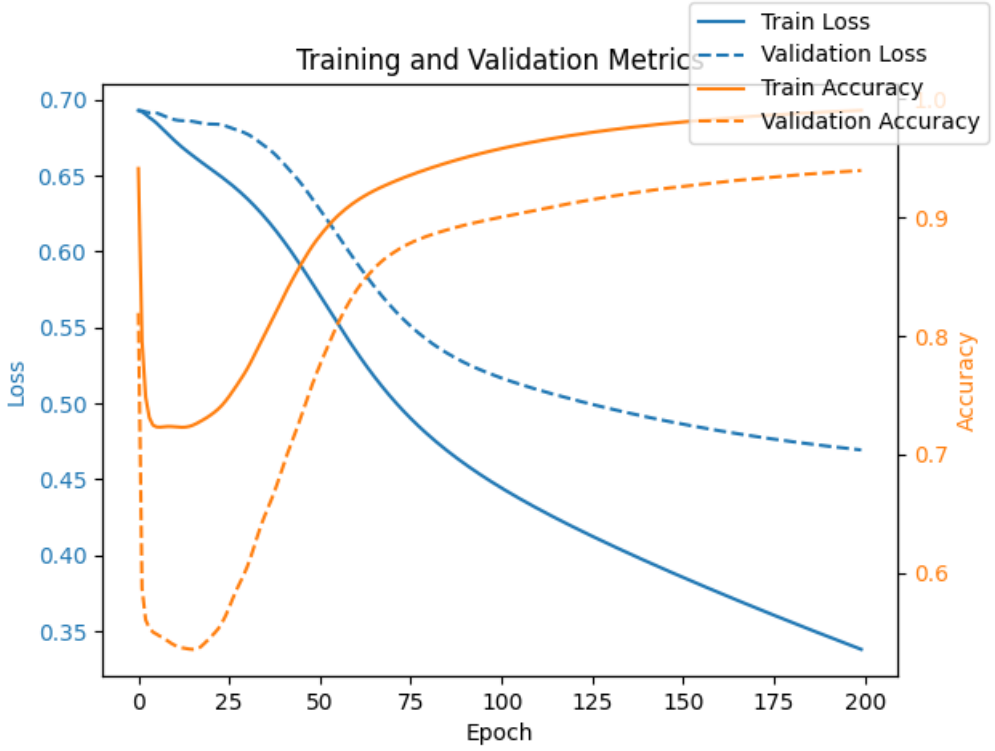
（1）cora数据集

笔者使用两层GCNLayer，激活函数为elu函数，使用自环，不使用dropedge以及pairnorm，epochs =200，hidden_size = 56， lr = 0.01， wd = 5e-4.

结果如下：

```
auc of the test set:0.9399320698229402
```

训练过程图像如下：



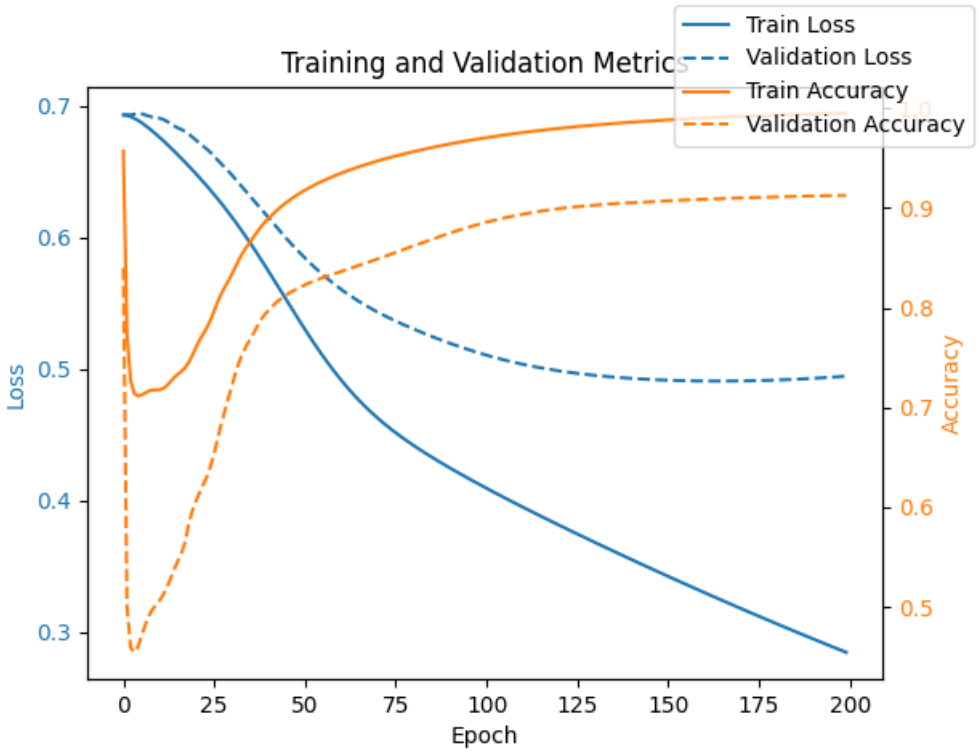
（2）citeseer数据集

笔者使用两层GCNLayer，激活函数为elu函数，使用自环，不使用dropedge以及pairnorm，epochs = 200，hidden_size = 56， lr = 0.01， wd = 5e-4.

结果如下：

```
auc of the test set:0.91270360787924
```

训练过程图像如下：



实验分析与总结

1. drop edge

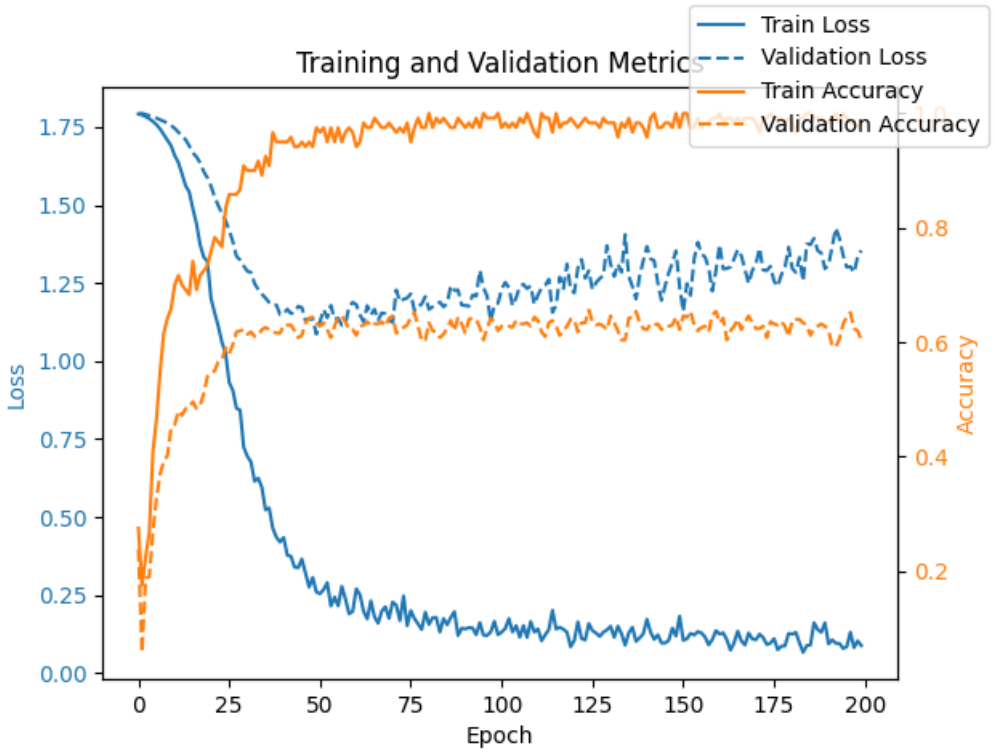
DropEdge指的是在每次训练时间内，随机剔除输入图的固定比率的边。将DropEdge应用于GCN可以增强了输入数据的随机性和多样性，缓解过拟合问题。在GCN中，相邻节点之间的消息传递是通过边路径传递的，删除一部分边会使得节点的链接更加稀疏，避免了GCN层数过多而导致的过平滑（梯度爆炸，梯度消失）的问题。

但在本实验中由于笔者在2层GCN的较为简单的网络架构下，已经可以训练出来较好的分数，所以最终都没有采用drop edge方法。而采用了drop edge方法之后准确率会有所下降，而且从训练函数图像可以看出，由于随机丢边导致非常不平稳，鲁棒性较差。

以在citeseer数据集上的节点预测为例，激活函数为relu函数，使用自环，使用dropedge，p=0.1，不使用pairnorm，3层GCNLayers， epochs =200，hidden_size = 32， lr = 0.01， wd = 5e-4.

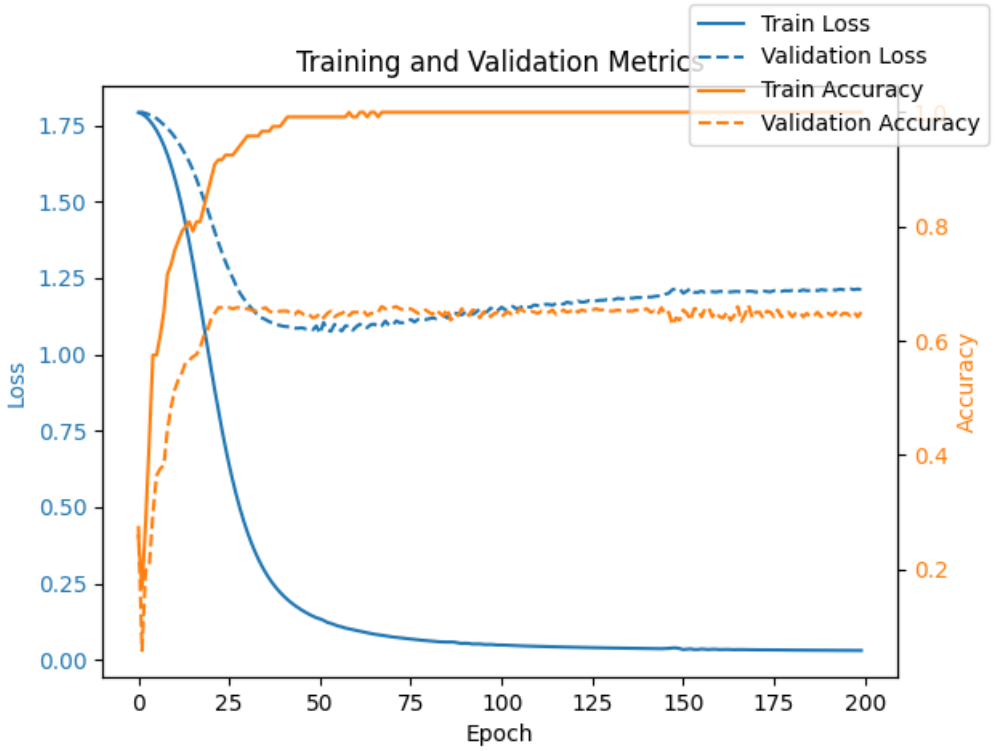
结果如下：

```
accuracy on test set:0.640999972820282
```



而不使用dropedge的3层3层GCNLayers的结果以及训练图像如下：

```
accuracy on test set:0.6639999747276306
```



2. pair norm

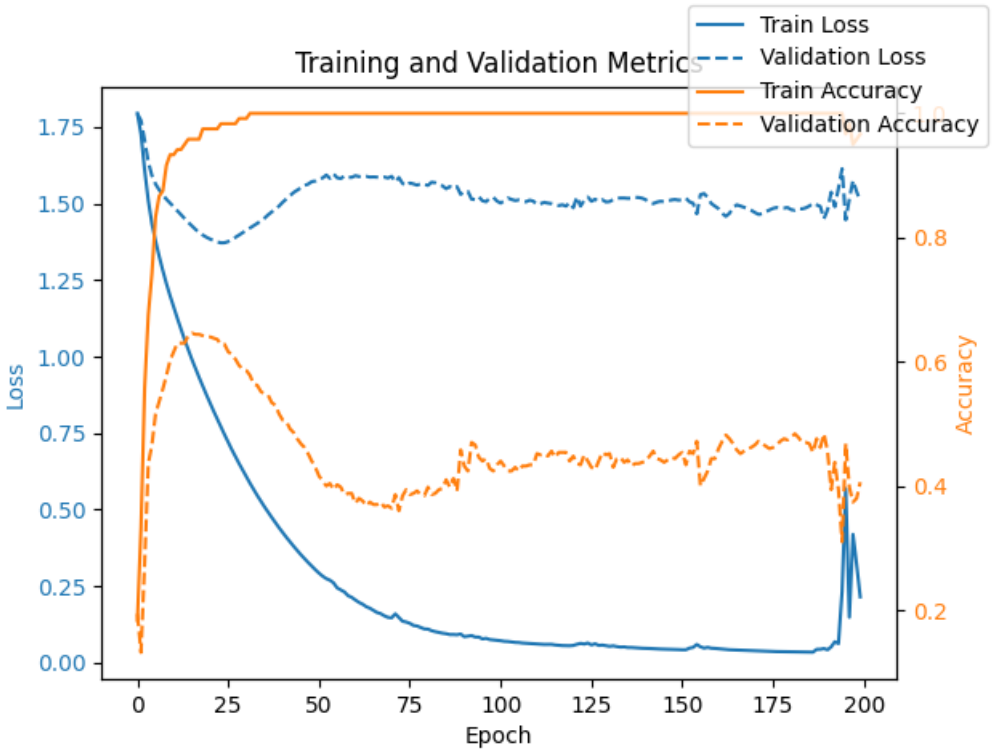
PairNorm会让距离较远的节点保持特征不相似，同时允许同一簇里的节点（相近的节点）特征有相似，同时尽可能保持正则化前后任何两个节点特征的总的距离保持不变。针对过平滑问题PairNorm可以每层GNN输出的所有节点特征之间的差异保持为一个常数，从而防止所有节点的特征趋于一致。

在本实验中由于笔者在2层GCN的较为简单的网络架构下，已经可以训练出来较好的分数，所以最终都没有采用pair norm 方法。在更深的网络中使用pair norm 方法, 从训练函数图像可以看出，非常不平稳，鲁棒性较差，且测试结果也较差。

以在citeseer数据集上的节点预测为例，激活函数为relu函数，使用自环，不使用dropedge，使用pairnorm，4层GCNLayers， epochs =200，hidden_size = 32， lr = 0.01， wd = 5e-4.

结果以及训练图像如下：

```
accuracy on test set:0.3959999978542328
```



3. self loop

self loop是自环，即在图上可以选择加上或者减去每个顶点自己到自己的边。在本实验中，每个任务都是加上自环会比不加自环的训练效果以及测试结果要好，因为首先加上自环会扩充训练数据。其次，只使用A的话，由于A的对角线上都是0，所以在和特征矩阵H相乘的时候，只会计算一个node的所有邻居的特征的加权和，该node自己的特征却被忽略了。因此，给A加上一个单位矩阵，即加上自环，这样就可以获得更多信息。

4. 激活函数

本次实验中笔者选用了relu、elu、leaky_relu等激活函数，在节点分类任务上没有较大不同，而在链路预测任务上的表现如下（此损失函数图像经平滑处理，5个epoch记录一次loss值）：

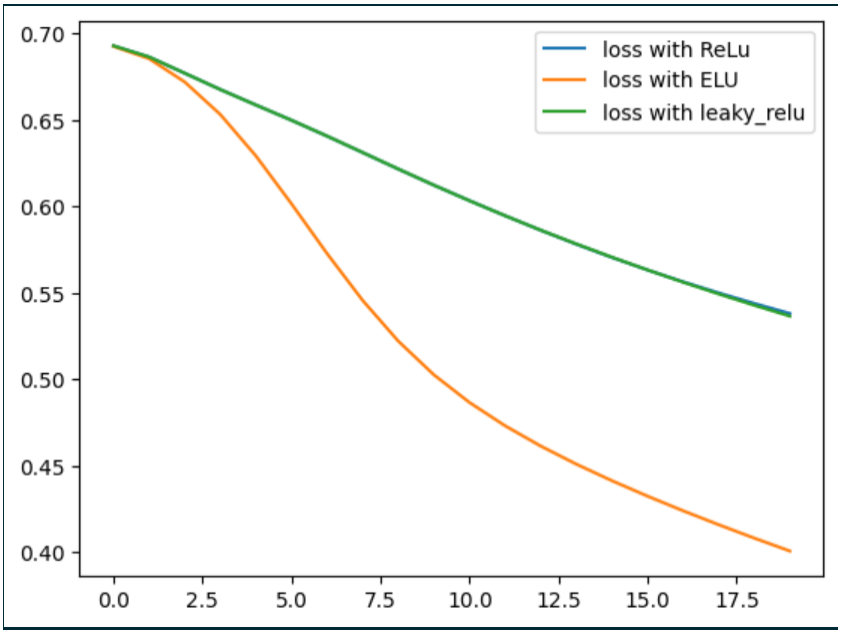


图1 cora数据集在链路预测上的训练损失随激活函数变化图像

5.网络深度

在本实验中，笔者对于两个数据集在两个任务上都选取了2层GCNLayers，加深网络层数都会导致模型的训练效果变差，可能是因为过拟合问题。

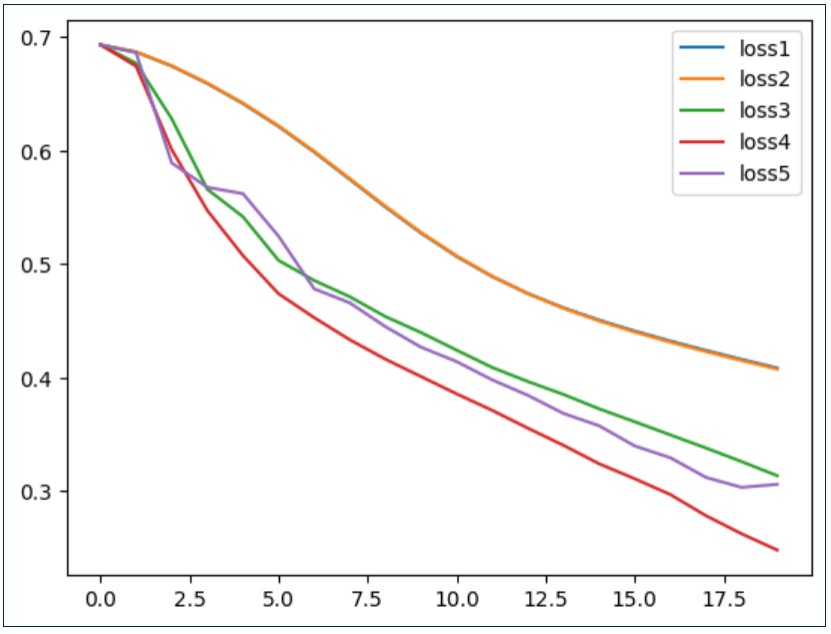


图2 cora数据集在链路预测上的训练损失随GCNLayers层数变化图像