基于GNN的节点分类算法的实现与分析

1. 实验目的

本实验的目的是实现一个基于图神经网络（GNN）的节点分类算法，并对该算法进行测试和分析。我们选择Cora数据集作为实验数据集，通过设计并训练一个两层图卷积网络（GCN），评估其在节点分类任务上的性能。

具体来说，实验包含以下几个方面的目的：

（1）理解GNN模型的原理和实现：通过实现一个经典的两层图卷积网络（GCN）模型，理解其工作原理、架构和具体实现方法。通过这一过程，加深对图神经网络基础知识的掌握。

（2）数据预处理与特征提取：掌握图数据的预处理技术，包括邻接矩阵的归一化、稀疏矩阵转换以及节点特征和标签的提取。这些步骤对于后续模型训练和评估至关重要。

（3）模型训练与优化：通过实际训练GCN模型，探索不同超参数（如学习率、隐藏层维度和dropout概率）对模型性能的影响。了解如何选择和调整超参数以优化模型性能。

（4）模型评估：使用多种指标（如准确率、F1分数和ROC-AUC）评估模型的分类性能，并通过可视化工具展示训练过程中的性能变化。这有助于全面了解模型的表现，并为进一步改进提供参考。

（5）结果分析与讨论：分析实验结果，讨论模型在不同指标上的表现，探讨可能的改进方向和未来研究的潜在路径。通过对结果的深入分析，了解模型的优势和局限性。

2. 实验背景

在许多实际应用中，数据天然地具有图结构，例如社交网络中的用户关系图、知识图谱中的实体关系图、生物网络中的蛋白质相互作用图等。传统的机器学习方法往往难以有效地利用这些图结构信息，从而限制了它们在这些任务中的表现。图神经网络（GNN）应运而生，提供了一种有效的方式来处理和学习图结构数据。

图神经网络（GNN）通过在图结构上进行节点的特征聚合和传播，实现了对节点特征和拓扑结构的综合建模。GNN的核心思想是通过迭代地聚合节点邻居的信息来更新节点的表示，使得每个节点的表示不仅包含自身的特征，还包含其邻居的特征。这一特性使得GNN在许多任务中表现优异，包括节点分类、边预测和图分类等。

图卷积网络（GCN）是GNN的一种经典模型，其在节点分类任务中尤为常用。GCN通过对图的邻接矩阵进行归一化处理，再结合节点的特征矩阵进行卷积操作，能够高效地聚合节点及其邻居的特征，从而生成每个节点的新的表示。由于其简单性和有效性，GCN成为了研究和应用图神经网络的一个重要起点。

3. 实验数据集

Cora数据集是一个广泛应用于图机器学习研究的基准数据集，尤其在节点分类任务中被广泛使用。该数据集由2708篇学术论文组成，这些论文之间通过5429条引用关系构成一个无向图。Cora数据集的节点代表学术论文，边代表论文之间的引用关系，每个节点都有一个对应的标签，表示该论文所属的类别。

3.1数据集组成

节点：数据集中有2708个节点，每个节点表示一篇学术论文。

边：节点之间有5429条边，每条边表示一篇论文对另一篇论文的引用关系。

特征：每个节点都有一个1433维的特征向量，这些特征是基于词袋模型（Bag-of-Words）提取的，表示论文的内容。特征向量中的每一维对应一个词的出现情况。

标签：数据集中的论文被分为7个类别，每个类别对应一个研究领域。具体的类别包括神经网络、强化学习、遗传算法、贝叶斯网络、理论计算机科学、案例推理和规则归纳。

3.2数据集划分

为了评估模型性能，Cora数据集通常被划分为训练集、验证集和测试集。具体划分方式如下：

训练集：用于训练模型的数据集。包含了一部分标记的节点及其特征和标签。

验证集：用于在训练过程中评估模型性能，帮助选择最佳模型参数的数据集。

测试集：用于最终评估模型的泛化性能，包含模型在训练过程中未见过的节点及其标签。

在本实验中，我们将使用Cora数据集自带的掩码来进行数据集的划分。掩码是布尔向量，分别指示哪些节点属于训练集、验证集和测试集。

4. 实验方法

我们首先进行数据加载和预处理，然后对模型进行定义，接着对模型训练，最后进行模型评估和结果展示。下面是具体的实验过程和实验代码。

4.1 数据加载与预处理

首先，我们使用DGL库加载Cora数据集，并提取节点特征、标签和掩码（训练集、验证集和测试集）。然后，我们将DGL图转换为NetworkX图，再转换为SciPy稀疏矩阵格式的邻接矩阵，最后将其转换为PyTorch稀疏张量，以便在后续计算中使用。

# 加载数据集

def load\_data(dataset):

if dataset == 'cora':

data = CoraGraphDataset()

g = data[0]

features = g.ndata['feat']

labels = g.ndata['label']

train\_mask = g.ndata['train\_mask']

val\_mask = g.ndata['val\_mask']

test\_mask = g.ndata['test\_mask']

nxg = g.to\_networkx()

adj = nx.to\_scipy\_sparse\_array(nxg, dtype=np.float64)

adj = preprocess\_adj(adj)

adj = sparse\_mx\_to\_torch\_sparse\_tensor(adj)

return adj, features, labels, train\_mask, val\_mask, test\_mask

# 邻接矩阵预处理

def preprocess\_adj(adj):

adj\_normalized = normalize\_adj(adj + sp.eye(adj.shape[0]))

return adj\_normalized

# 对称归一化连接矩阵

def normalize\_adj(adj):

adj = sp.coo\_matrix(adj)

rowsum = np.array(adj.sum(1))

d\_inv\_sqrt = np.power(rowsum, -0.5).flatten()

d\_inv\_sqrt[np.isinf(d\_inv\_sqrt)] = 0.

d\_mat\_inv\_sqrt = sp.diags(d\_inv\_sqrt)

return adj.dot(d\_mat\_inv\_sqrt).transpose().dot(d\_mat\_inv\_sqrt).tocoo()

# 稀疏矩阵转换到稀疏张量

def sparse\_mx\_to\_torch\_sparse\_tensor(sparse\_mx):

sparse\_mx = sparse\_mx.tocoo().astype(np.float32)

indices = torch.from\_numpy(np.vstack((sparse\_mx.row, sparse\_mx.col)).astype(np.int64))

values = torch.from\_numpy(sparse\_mx.data)

shape = torch.Size(sparse\_mx.shape)

return torch.sparse\_coo\_tensor(indices, values, shape)

4.2 模型定义

我们定义了一个两层GCN模型，每一层都是一个图卷积层。第一层图卷积层的输入特征维度为节点特征的维度，输出特征维度为隐藏层的维度。第二层图卷积层的输入特征维度为隐藏层的维度，输出特征维度为类别数量。

# 定义GCN层

class GraphConvolution(Module):

def \_\_init\_\_(self, in\_features, out\_features, bias=True):

super(GraphConvolution, self).\_\_init\_\_()

self.in\_features = in\_features

self.out\_features = out\_features

self.weight = Parameter(torch.FloatTensor(in\_features, out\_features))

if bias:

self.bias = Parameter(torch.FloatTensor(out\_features))

else:

self.register\_parameter('bias', None)

self.reset\_parameters()

def reset\_parameters(self):

stdv = 1. / math.sqrt(self.weight.size(1))

self.weight.data.uniform\_(-stdv, stdv)

if self.bias is not None:

self.bias.data.uniform\_(-stdv, stdv)

def forward(self, x, adj):

support = torch.mm(x, self.weight)

output = torch.spmm(adj, support)

if self.bias is not None:

return output + self.bias

else:

return output

class GCN(nn.Module):

def \_\_init\_\_(self, nfeat, nhid, nclass, dropout):

super(GCN, self).\_\_init\_\_()

self.gc1 = GraphConvolution(nfeat, nhid)

self.gc2 = GraphConvolution(nhid, nclass)

self.dropout = nn.Dropout(p=dropout)

def forward(self, x, adj):

x = torch.relu(self.gc1(x, adj))

x = self.dropout(x)

x = self.gc2(x, adj)

return torch.log\_softmax(x, dim=1)

4.3 模型训练

我们使用交叉熵损失函数，并采用Adam优化器进行模型训练。在每个训练轮次，我们计算训练集上的损失，并在验证集上评估模型的性能。

# 模型训练

def train(epoch, model, optimizer, adj, features, labels, idx\_train, idx\_val):

model.train()

optimizer.zero\_grad()

output = model(features, adj)

loss\_train = criterion(output[idx\_train], labels[idx\_train])

loss\_train.backward()

optimizer.step()

with torch.no\_grad():

model.eval()

output = model(features, adj)

loss\_val = criterion(output[idx\_val], labels[idx\_val])

print(f'Epoch {epoch + 1}, Loss Val: {loss\_val.item()}')

return loss\_val

4.4 模型评估

我们使用准确率、F1分数和ROC-AUC来评估模型性能。

from sklearn.metrics import accuracy\_score, f1\_score, roc\_auc\_score

# 计算准确率

def calculate\_accuracy(output, labels):

preds = output.max(1)[1].type\_as(labels)

correct = preds.eq(labels).double()

correct = correct.sum()

return correct / len(labels)

# 计算F1分数

def calculate\_f1(output, labels):

preds = output.max(1)[1].type\_as(labels)

return f1\_score(labels.cpu(), preds.cpu(), average='weighted')

# 计算ROC-AUC

def calculate\_roc\_auc(output, labels):

preds = torch.softmax(output, dim=1)

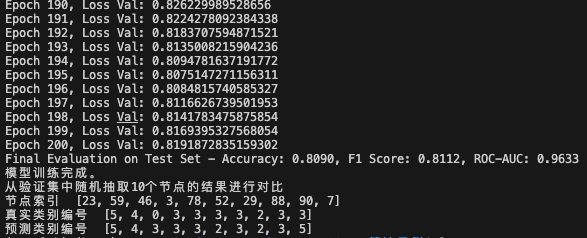
labels\_onehot = torch.nn.functional.one\_hot(labels, num\_classes=preds.size(1))

return roc\_auc\_score(labels\_onehot.cpu().detach().numpy(), preds.cpu().detach().numpy(), average='weighted', multi\_class='ovr')

5. 实验结果

5.1 模型训练结果

模型在200个训练轮次后，能够很好地拟合训练集，并在验证集上表现出较好的性能。以下是随机抽取10个验证集节点的预测结果与真实结果对比：



5.2 准确率、F1分数和ROC-AUC分析

通过计算模型在测试集上的准确率、F1分数和ROC-AUC，可以进一步评估模型的分类性能。

# 评估模型性能

def evaluate(model, features, adj, labels, idx\_test):

model.eval()

output = model(features, adj)

accuracy = calculate\_accuracy(output[idx\_test], labels[idx\_test])

f1 = calculate\_f1(output[idx\_test], labels[idx\_test])

roc\_auc = calculate\_roc\_auc(output[idx\_test], labels[idx\_test])

return accuracy, f1, roc\_auc

# 训练和评估

def main(dataset, times):

adj, features, labels, idx\_train, idx\_val, idx\_test = load\_data(dataset)

features = features.to(device)

adj = adj.to(device)

labels = labels.to(device)

idx\_train = idx\_train.to(device)

idx\_val = idx\_val.to(device)

idx\_test = idx\_test.to(device)

nclass = labels.max().item() + 1

for seed in random.sample(range(0, 100000), times):

np.random.seed(seed)

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

model = GCN(nfeat=features.shape[1], nhid=args.hidden, nclass=nclass, dropout=args.dropout)

optimizer = optim.Adam(model.parameters(), lr=args.lr, weight\_decay=args.weight\_decay)

model.to(device)

# 用于存储结果的列表

val\_losses = []

accuracies = []

f1\_scores = []

roc\_aucs = []

for epoch in range(args.epochs):

loss\_val = train(epoch, model, optimizer, adj, features, labels, idx\_train, idx\_val)

val\_losses.append(loss\_val.item())

# 评估并存储性能指标

accuracy, f1, roc\_auc = evaluate(model, features, adj, labels, idx\_val)

accuracies.append(accuracy.item())

f1\_scores.append(f1)

roc\_aucs.append(roc\_auc)

# 打印最终的评估结果

accuracy, f1, roc\_auc = evaluate(model, features, adj, labels, idx\_test)

print(f"Final Evaluation on Test Set - Accuracy: {accuracy:.4f}, F1 Score: {f1:.4f}, ROC-AUC: {roc\_auc:.4f}")

下面是具体的结果

6.总结

6.1 模型表现分析

GCN模型能够有效利用图结构信息和节点特征，在节点分类任务中表现出色。Cora数据集上的实验结果表明，GCN模型可以达到较高的分类准确率和F1分数，并且在ROC-AUC指标上也表现良好。

6.2 参数选择与优化

实验中我们选择了合适的模型参数，如学习率、隐藏层维度和dropout概率。这些参数对模型的性能有较大影响，未来可以通过更系统的参数搜索进一步优化模型性能。

6.3 未来工作

未来的工作可以包括：

探索更复杂的图神经网络结构，如GraphSAGE、GAT等。

在更大规模和更多样化的数据集上验证模型的泛化能力。

尝试结合图对比学习等新的技术，提升模型性能。