**LightGCN代码复现**

1. **Loader**

以训练数据加载函数train\_loader(file\_path)为例，具体实现说明，如下所示。

1. 按行读取指定路径的train.txt文本数据，并返回训练用户-物品交互字典user\_item\_dict，即：{user\_id:[item\_id, ...], ...}。
2. 根据前述生成的user\_item\_dict，获取总用户数num\_users、总物品数num\_items以及对称邻接矩阵A，即：A=[[0,R],[R^T,0]]，R根据user\_item\_dict转换而来，A的维度为(num\_users+num\_items)\*(num\_users+num\_items)。

# 6. 构建对称邻接矩阵A=[[0, R], [R^T, 0]]  
top\_row = torch.cat((torch.zeros((R.size(0), R.size(0))), R), dim=1)  
bottom\_row = torch.cat((R.t(), torch.zeros((R.size(1), R.size(1)))), dim=1)  
A = torch.cat((top\_row, bottom\_row), dim=0)

1. 计算矩阵A的度矩阵D，以完成对称邻接矩阵A的L2归一化处理，从而返回归一化后的对称邻接矩阵A\_normalized，即：。

值得注意的是，使用构造而来的对称邻接矩阵A作为卷积聚合过程中的图，而非直接使用维度为num\_users\*num\_items的R，是考虑通过构建一个双向且对称的用户-物品关系图，能够更灵活地学习用户和物品方向上的节点间关系。

D = torch.sum(A, dim=1) # 7. 计算A的度矩阵D  
D\_sqrt\_inv = torch.diag(torch.pow(D, -0.5))  
A\_normalized = torch.mm(torch.mm(D\_sqrt\_inv, A), D\_sqrt\_inv) # 8. L2归一化处理

1. 从user\_item\_dict中随机采样train\_inter\_num个样本，用于LightGCN模型的迭代训练，即：samples[i]=[user\_id, pos\_item\_id, neg\_item\_id]。

# 10. 样本采样  
users = np.random.randint(0, num\_users, train\_inter\_num); samples = []  
for user in users:  
 pos\_item\_list = user\_item\_dict[user]  
 if len(pos\_item\_list) == 0: continue  
 pos\_index = np.random.randint(0, len(pos\_item\_list))  
 pos\_item = pos\_item\_list[pos\_index]  
 while True:  
 neg\_item = np.random.randint(0, num\_items)  
 if neg\_item in pos\_item\_list:  
 continue  
 else:  
 break  
 samples.append([user, pos\_item, neg\_item])

1. **LightGCN**

LightGCN模型的实现方法，具体如下所示。其中，**可学习参数**仅有用户嵌入层权重users\_embedding.weight和物品嵌入层权重items\_embedding.weight，它们在初始化方法中需预先定义并将权重取值进行随机初始化。在前向传播函数forward中，需将用户和物品嵌入权重按行**拼接**以获得初始的嵌入结果emb，用于匹配同adj\_mat作矩阵乘法所需的维度要求，即：在卷积聚合运算的过程中，将维度为(num\_users+num\_items)\*(num\_users+num\_items)的图矩阵adj\_mat同维度为(num\_users+num\_items)\*embedding\_dim的emb相乘，以获得不同层次的嵌入结果emb（维度保持不变）。此外，在每次卷积聚合计算之前，可以使用F.dropout处理L2归一化后的对称邻接矩阵adj\_mat，从而引入**dropout**手段用于处理过拟合问题。

值得注意的是，LightGCN本质上同神经协同过滤算法一致，都需根据用户的历史交互物品和物品的历史交互用户，来学习不同用户和物品在潜在因素空间中的具体表征，并通过目标用户和不同物品间的点积结果来衡量相似性，从而完成基于近邻的Top-N推荐任务。**不同之处在于**，LightGCN递归地将当前的嵌入结果emb同图矩阵相乘，从而获得了不同抽象层次的嵌入结果（比如，第二层生成的用户表征，是根据具有交互项目重叠的全部用户的平滑结果）。令人瞩目的是，为应对深层卷积聚合过程中的过平滑问题，原文作者直接通过加和平均的方式，完成了各层嵌入结果的**嵌入组合**。此外，从上周的实验来看，使用注意力机制学习不同抽象层次的嵌入结果在最终表征中的参考程度，实际效果反而不如这种不需要引入额外复杂性的加和平均方式。

class LightGCN(nn.Module):def \_\_init\_\_(self, users\_num, items\_num, embedding\_dim, layers, keep\_prob, adj\_mat):  
 super(LightGCN, self).\_\_init\_\_()  
 self.users\_num = users\_num  
 self.items\_num = items\_num  
 self.embedding\_dim = embedding\_dim  
 self.layers = layers  
 self.keep\_prob = keep\_prob  
 self.adj\_mat = adj\_mat

self.users\_embedding = nn.Embedding(users\_num, embedding\_dim)  
 self.items\_embedding = nn.Embedding(items\_num, embedding\_dim)  
 self.reset\_parameters()  
 self.sigmoid = nn.Sigmoid()  
 def reset\_parameters(self):nn.init.normal\_(self.users\_embedding.weight, std=0.1)  
 nn.init.normal\_(self.items\_embedding.weight, std=0.1)  
 def forward(self):  
 dense\_adj\_mat = self.adj\_mat.to\_dense()  
 user\_emb = self.users\_embedding.weight  
 item\_emb = self.items\_embedding.weight  
 emb = torch.cat([user\_emb, item\_emb], dim=0)  
 embs = [emb]  
 for \_ in range(self.layers):  
 dense\_adj\_mat = F.dropout(dense\_adj\_mat, p=self.keep\_prob, training=self.training)  
 emb = sparse.mm(dense\_adj\_mat, emb)  
 embs.append(emb)  
 final\_emb = torch.stack(embs, dim=1)  
 final\_emb = torch.mean(final\_emb, dim=1)  
 users\_token, items\_token = final\_emb[:self.users\_num], final\_emb[self.users\_num:]  
 return users\_token, items\_token  
 def get\_user\_rating(self, user\_indices):users\_token, items\_token = self.forward()  
 target\_users\_emb, items\_emb = users\_token[user\_indices.long()], items\_token  
 scores = self.sigmoid(torch.matmul(target\_users\_emb, items\_emb.t()))  
 return scores

1. **train\_pairwise**

train\_pairwise()的实现方式较为简单，仅需打乱样本顺序并分批次进行迭代训练即可。关键在于，如何使用自定义的BPRLoss损失函数，完成损失的计算，继而进行梯度更新。BPRLoss本质上是通过对比样本中正、负例物品偏好分数的方式来计算损失，同时根据批次用户和对应正、负例物品的嵌入来计算正则化损失reg\_loss，从而用于应对过拟合问题。

class BPRLoss(object):  
 def \_\_init\_\_(self, weight\_decay):  
 self.weight\_decay = weight\_decay  
 def \_\_call\_\_(self,users\_embeddings,pos\_items\_embeddings,neg\_items\_embeddings,  
 users\_embeddings\_layer0,pos\_items\_embeddings\_layer0,neg\_items\_embeddings\_layer0,

parameters\_norm):  
 reg\_loss = (1 / 2) \* (users\_embeddings\_layer0.norm(2).pow(2) +  
 pos\_items\_embeddings\_layer0.norm(2).pow(2) +  
 neg\_items\_embeddings\_layer0.norm(2).pow(2) + parameters\_norm  
 ) / users\_embeddings.shape[0] # 计算正则化损失  
 pos\_scores = torch.mul(users\_embeddings, pos\_items\_embeddings) # 计算正例物品分数  
 pos\_scores = torch.sum(pos\_scores, dim=1)  
 neg\_scores = torch.mul(users\_embeddings, neg\_items\_embeddings) # 计算负例物品分数  
 neg\_scores = torch.sum(neg\_scores, dim=1)  
 loss = torch.mean(torch.nn.functional.softplus(neg\_scores - pos\_scores)) # 以差值作为损失  
 reg\_loss \*= self.weight\_decay # 将正则化损失乘以权重衰减系数，并加到总损失上。  
 loss += reg\_loss  
 return loss

1. **test\_pairwise**

每迭代训练10个周期，即可使用测试集对模型性能进行综合评估，具体如下所示。

1. 根据model.get\_user\_rating()，获取批次用户的预测结果rating，即：rating = [[item1, item2, ...], ...]，尺寸为test\_batch\_size\*num\_item。
2. 根据测试用户-物品交互字典test\_dict，获取批次用户的标签结果label，即：label= [[pos\_item1, ...], ...]，尺寸为test\_batch\_size\*不定长。不定长是因为每个用户在测试集的真实交互物品数不同。
3. 根据测试用户索引及其在训练集中出现过的交互物品索引，将rating中相应位置处的预测得分设置为很小的值，从而在后续的Top-K推荐列表中排除训练物品。具体如下所示。

值得注意的是，对于每个用户而言，训练和测试集中的交互物品不同。而且，在测试集基础上计算评估度量的过程中，主要关注测试集中用户交互物品的被推荐程度。然而，对于使用训练集学习而来的推荐模型而言，它更可能会对每个测试用户推荐其在训练集中出现过的交互物品。因此，在从rating中抽取每个测试用户的Top-K推荐结果之前，需要将对应的训练交互物品分数设置为很小的值。实验证明，这种特殊的处理方式，能够一定程度上提高推荐模型在测试集上的性能表现。

batch\_users\_pos, exclude\_index, exclude\_items = [], [], []  
for user in batch\_users:  
 batch\_users\_pos.append(train\_dict[user])  
for range\_i, items in enumerate(batch\_users\_pos):  
 exclude\_index.extend([range\_i] \* len(items))  
 exclude\_items.extend(items)  
rating[exclude\_index, exclude\_items] = -(1 << 10)

1. 使用torch.topk(rating, k=topk)方法，获取每个测试用户的前K个最高评分物品，即：rating\_K=[[pretop1\_item\_id, pretop2\_item\_id, ...], ...]，尺寸为test\_batch\_size\*topk。
2. 根据所有批次测试用户的rating\_K以及label，计算并返回最终的平均准确率precision、平均召回率recall以及平均NDCG，具体如下所示。

users\_list.append(batch\_users)  
 rating\_list.append(rating\_K.cpu())  
 ground\_truth\_list.append(label)  
X = zip(rating\_list, ground\_truth\_list)  
precision, recall, ndcg = [], [], []  
for batch, x in enumerate(X):  
 rec, pre, ndc = test\_one\_batch(x, topk)  
 precision.append(pre); recall.append(rec); ndcg.append(ndc)  
return np.mean(precision), np.mean(recall), np.mean(ndcg)

1. **main**

主函数的实现方式，具体如下所示。其中，需事先设定随机数种子，从而保证实验结果的可重复性。同时，还需根据上周实验的经验，完成若干超参数的取值。

def main():  
 # 初始设置  
 seed = 42  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed\_all(seed)  
 train\_data\_path = "amazon-electro/train.txt"  
 test\_data\_path = "amazon-electro/test.txt"  
 device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  
 # 参数定义  
 topk = 20  
 layers = 4  
 epochs = 500  
 keep\_prob = 0.1  
 embedding\_dim = 64  
 batch\_size = 2048  
 test\_batch\_size = 100  
 lr = 0.001  
 weight\_decay = 1e-4  
 # 1. 加载数据  
 adj\_mat, samples, users\_num, items\_num, train\_dict = Loader.train\_loader(train\_data\_path)  
 test\_dict = Loader.test\_loader(test\_data\_path)  
 adj\_mat = adj\_mat.to(device)  
 # 2. 加载模型  
 model = lightgcn.LightGCN(users\_num, items\_num, embedding\_dim, layers, keep\_prob, adj\_mat)  
 model.to(device)  
 # 3. 加载损失函数和优化器  
 loss\_fun = BPRLoss.BPRLoss(weight\_decay=weight\_decay)  
 optimizer = torch.optim.Adam(model.parameters(), lr=lr)  
 # 4. 模型训练与测试  
 for epoch in range(epochs):  
 avg\_loss = train\_pairwise.train\_pairwise(samples, model, loss\_fun, optimizer, batch\_size, device)  
 if epoch % 10 == 0 and epoch != 0:  
 precision, recall, ndcg = eval\_pairwise.eval\_pairwise(train\_dict, test\_dict, model, test\_batch\_size, topk, device)  
 # 5. 结果可视化

本次实验选用小型数据集amazon-electro进行LightGCN模型的迭代训练和测试。具体实验结果，如下图所示。关于结果可视化的实现方式，此处省略。

