1. **代码实现**
2. **数据预处理**

由于本次实验，使用了gowalla、yelp2018及amazon-movies等数据集。因此，需根据训练及测试数据文件，制定统一的预处理范式，以获得训练用户-项目交互矩阵、训练用户分箱字典以及测试用户交互字典。实现方式，具体如下所示。

即，在Loader实现类的初始化方法中：

1. 遍历训练数据文件**train.txt**中的每个行样本，并通过切片的方式将样本中的用户ID追加存储到非重训练用户列表train\_unique\_users中，将样本中的所有交互项目ID追加存储到训练用户交互列表train\_item中，往训练用户列表train\_user中追加存储同样本交互项目等量的重复用户ID元素（保证一维列表train\_item和train\_user中对应索引处的用户和交互项目一一对应）。同时，还需以样本的用户ID为键，以包含样本所有交互项目ID的列表为值，追加存储到训练用户交互字典user\_interactions\_dict\_train中。此外，还需根据样本中的最大交互项目和用户ID，更新最大项目数self.m\_item和最大用户数self.n\_user的取值。
2. 重复上述方法，遍历测试数据文件**test.txt**中的每个行样本，以获得非重测试用户列表test\_unique\_users、测试用户列表test\_user、测试用户交互列表test\_item、测试用户交互字典\_\_testDict。同样，也需要更新最大项目数self.m\_item和最大用户数self.n\_user的默认取值（为考虑后续构建交互矩阵的用户和项目维数，需要知道最大的用户和项目ID）。
3. 使用scipy.sparse.csr\_matrix()，构建训练用户-项目稀疏矩阵user\_item\_net。实现方式，如下所示。若存在交互（用户0，项目1）（用户1，项目2）（用户1，项目3），则user\_item\_net取值为：[[项目1], [项目2, 项目3], ]。

self.user\_item\_net = csr\_matrix((  
 np.ones(len(self.train\_user)), # 元素取值，取值为1且等长train\_user的数组  
 (self.train\_user, self.train\_item)), # 元素位置，（user\_id, item\_id）  
 shape=(self.n\_user, self.m\_item)) # 矩阵形状

（4）基于配置参数num\_bins与训练用户交互字典，根据交互项目数将训练用户分配到不同区间中，用于考虑推荐的公平性问题。实现方式，如下所示。最终返回一个将用户映射到其所属箱子索引的字典user\_bin\_dict，即：{user\_id: bin\_index, ...}。

def distribute\_users\_into\_bins\_by\_num\_interactions(self, num\_bins):  
 # 获得每个训练用户的对数化交互项目数  
 log\_values = [np.log(len(self.user\_interactions\_dict\_train[user])) for user in self.user\_interactions\_dict\_train.keys()]  
 min\_num\_interactions = min(log\_values) # 对数化后的最小交互物品数  
 max\_num\_interactions = max(log\_values) # 对数化后的最大交互物品数  
 # 在最小最大值间生成num\_bins个均匀间隔的阈值，用于将用户分配到不同的箱子中  
 bin\_thresholds = np.linspace(min\_num\_interactions, max\_num\_interactions, num\_bins)  
 # 将对数化后的交互物品数量根据阈值分配到不同的箱子中  
 bin\_indices = np.digitize(log\_values, bin\_thresholds, right=True)  
 # 创建一个将用户映射到其所属箱子索引的字典，{user\_id: bin\_index, }  
 user\_bin\_dict = dict(zip(self.user\_interactions\_dict\_train.keys(), bin\_indices))  
 return user\_bin\_dict

1. **模型定义**

**2.1 PPR模型**

基于幂迭代的Personalized PageRank计算模块，具体实现方式，如下所示。

在PPR的**初始化**中。首先，将表示随机游走概率的参数alpha和幂迭代次数的参数niter，保存为类属性。然后，使用自定义的calc\_A\_hat函数，归一化处理用户-项目交互邻接矩阵。再后，将计算得到的（1-alpha）\*M 转换为pytorch稀疏张量A\_hat。最后，初始化自定义的MixedDropout方法，可用于对稀疏矩阵的节点进行随机置零失活处理。

在PPR的**前向传播**中。首先，接收归一化邻接矩阵E，作为预测矩阵preds的默认取值。然后，使用dropout处理后的的A\_drop，根据，迭代更新出最后学习到的嵌入矩阵preds。

class PPRPowerIteration(nn.Module):def \_\_init\_\_(self, adj\_matrix: sp.spmatrix, alpha: float, niter: int, drop\_prob: float = None):  
 super().\_\_init\_\_()  
 self.alpha = alpha  
 self.niter = niter  
 M = calc\_A\_hat(adj\_matrix) # 获取归一化邻接矩阵M  
 self.register\_buffer('A\_hat', sparse\_matrix\_to\_torch((1 - alpha) \* M)) # 获取A\_hat  
 if drop\_prob is None or drop\_prob == 0:  
 self.dropout = lambda x: x  
 else:  
 self.dropout = MixedDropout(drop\_prob)  
 def forward(self, E: torch.FloatTensor):  
 preds = E # 将E作为预测矩阵的起始值  
 for \_ in range(self.niter): # 进行niter次迭代  
 A\_drop = self.dropout(self.A\_hat) # 对A\_hat进行dropout处理  
 preds = A\_drop @ preds + self.alpha \* E  
 return preds

**2.2 LightGCN模型**

本节主要考虑LightGCN模型的基本实现方式，具体解释，如下所示。

在LightGCN的**初始化**中。首先，需根据输入的配置参数，初始化num\_users、num\_items、latent\_dim、n\_layers、keep\_prob及a\_split。然后，使用torch.nn.Embedding函数，分别创建用户和项目嵌入层。再后，若配置参数中未声明使用预训练数据，可直接根据正态分布随机初始化嵌入权重；否则，则需从配置参数中加载预训练的用户和项目嵌入权重。最后，还需初始化：Sigmoid激活函数、关于获取数据集的稀疏图表示方法、空的嵌入列表embs。

在LightGCN的**前向传播**中。首先，将用户和项目的嵌入权重拼接起来，获得维数为(num\_users+num\_items)\*latent\_dim的初始嵌入权重向量all\_emb，并存储到嵌入列表embs中。其次，若配置参数中声明了需要启用dropout操作，则在模型的训练阶段完成图中节点的随机失活（图中的节点指代稀疏邻接矩阵中的行）。然后，根据事先指定的卷积聚合次数n\_layers，迭代地将前层嵌入权重all\_emb同归一化处理后的稀疏邻接矩阵相乘，从而通过当前层的嵌入结果更新all\_emb，并将新的嵌入向量all\_emb追加存储到嵌入列表embs中。再后，可根据嵌入列表embs中来自不同卷积聚合层次的嵌入向量，通过直接求均值的方式完成嵌入组合环节，从而最终获得所有用户和项目的嵌入表示light\_out。最后，将维数为(num\_users+num\_items)\*latent\_dim的嵌入结果light\_out拆分并返回用户和项目嵌入矩阵。

**2.3 引入注意力考虑：嵌入组合**

一种想法是，可以考虑引入注意力机制来学习嵌入组合权重，从而代替原文通过求均值的方式组合不同层次的嵌入结果all\_emb。具体实现方式，如下所示。如若LightGCN存在4个图卷积层，则注意力分数attention\_scores维度为（1, 4+1, 1），从而用于考虑来自4个图卷积层的嵌入结果和初始嵌入，在最终嵌入矩阵light\_out中的参考程度。

# 在初始化方法中，额外创建一个可训练参数attention\_weights

self.attention\_weights = torch.nn.Parameter(torch.randn(config["n\_layers"]+1), requires\_grad=True)

# 在前向传播函数中，使用注意力权重代替嵌入组合权重a\_k  
attention\_scores = F.softmax(self.attention\_weights, dim=0) # attention\_scores\_size:(1, n\_layers+1, 1)  
light\_out = torch.sum(embs \* attention\_scores.view(1, -1, 1), dim=1)

**2.4 引入用户和项目注意力考虑：嵌入组合**

更进一步，也可通过考虑不同用户和项目注意力分数的方式，来对不同层次的嵌入结果进行加权求和。具体实现方式，如下所示。其中，注意力分数矩阵attention\_scores的维度为（num\_users+num\_items, n\_layers+1, 1），嵌入结果列表embs的维度为（num\_users+num\_items, latent\_dim, n\_layers+1）。

# 在初始化方法中，额外创建可训练参数attention\_weights\_users及attention\_weights\_items,size=(n\_layers+1,)

self.attention\_weights\_users = torch.nn.Parameter(torch.randn(config["n\_layers"]+1),requires\_grad=True)  
self.attention\_weights\_items = torch.nn.Parameter(torch.randn(config["\_layers"]+1),requires\_grad=True)

# 通过softmax标准化注意力权重的取值，并重塑尺寸为(1, n\_layers+1, 1)  
attention\_scores\_users = F.softmax(self.attention\_weights\_users, dim=0).view(1, -1, 1)  
attention\_scores\_items = F.softmax(self.attention\_weights\_items, dim=0).view(1, -1, 1)  
# 拼接用户和项目的注意力分数，注意力分数的尺寸为（num\_users + num\_items, n\_layers + 1, 1）  
attention\_scores = torch.cat([  
 attention\_scores\_users.repeat(self.num\_users, 1, 1),  
 attention\_scores\_items.repeat(self.num\_items, 1, 1)], dim=0)  
# 计算加权和，将嵌入向量与注意力分数相乘，并在特定维度上进行求和。  
light\_out = torch.sum(embs \* attention\_scores, dim=1)

**2.5 引入缩放点积注意力代替：嵌入结果列表embs**

另一种想法是，可以考虑使用缩放点积注意力矩阵代替嵌入结果列表embs，继而完成嵌入组合工作。具体实现方式，如下所示。其中，引入注意力的嵌入结果矩阵attention\_output是依赖于取值皆为embs的Q、K、V，通过缩放点积的方式计算而来。

@staticmethod  
def compute\_attention(queries, keys, values):# 计算缩放因子，即：查询向量Q维度的平方根  
 scaling\_factor = math.sqrt(queries.size(-1))  
 # 计算注意力分数，即：Q与K点积，然后除以缩放因子  
 attention\_scores = torch.matmul(queries, keys.transpose(-2, -1)) / scaling\_factor  
 # 将注意力分数应用softmax函数，得到注意力权重  
 attention\_weights = F.softmax(attention\_scores, dim=-1)  
 # 使用注意力权重对值向量进行加权求和，得到最终的注意力输出。  
 attention\_output = torch.matmul(attention\_weights, values)  
 return attention\_output

queries, keys, values = self.prepare\_attention\_inputs(embs) # q=k=v=embs  
attention\_output = self.compute\_attention(queries, keys, values) # 计算缩放点积注意力矩阵

light\_out = torch.mean(attention\_output, dim=1) # 直接求均值，完成嵌入组合

**2.6 引入带有线性投影的缩放点积注意力代替：嵌入结果列表embs**

更进一步，也可以通过引入线性投影层的方式，分别将Q、K、V的嵌入维latent\_dim，灵活独立地映射到维数为attention\_dim的查询、键和值空间中去。具体实现方式，如下所示。

# 在初始化方法中：创建三个线性投影层，分别用于将嵌入向量投影到查询、键和值空间  
self.query\_projection = nn.Linear(self.latent\_dim, self.attention\_dim, bias=False)  
self.key\_projection = nn.Linear(self.latent\_dim, self.attention\_dim, bias=False)  
self.value\_projection = nn.Linear(self.latent\_dim, self.attention\_dim, bias=False)

def prepare\_attention\_inputs(self, embs):  
 queries = self.query\_projection(embs)  
 keys = self.key\_projection(embs)  
 values = self.value\_projection(embs)  
 return queries, keys, values

# 对embs的嵌入维分别进行查询、键和值空间投影，从而得到q k v

queries, keys, values = self.prepare\_attention\_inputs(embs)

attention\_output = self.compute\_attention(queries, keys, values) # 计算缩放点积注意力矩阵

light\_out = torch.mean(attention\_output, dim=1) # 直接求均值，完成嵌入组合

1. **损失函数与优化器**

本次实验，选择使用**BPRLoss**作为损失函数，具体实现方式，如下所示。其中，首先使用L2范数对每个用户、正例和负例项目的嵌入向量进行正则化，并除以嵌入向量的数量以便损失不受批量大小的影响，从而完成正则化损失reg\_loss的计算。其次，分别计算用户嵌入与正、负例项目嵌入间的点积得分，并求和。然后，使用softplus函数处理正例得分pos\_scores与负例得分neg\_scores间的差值，确保差值始终为正，从而通过求平均值的方式正确计算出损失loss。最后，将正则化损失reg\_loss乘以权重衰减系数weight\_decay，累加到总损失上，即可得出最终的总损失。

class BPRLoss(object):def \_\_init\_\_(self, weight\_decay):self.weight\_decay = weight\_decay  
 def \_\_call\_\_(self,users\_embeddings,pos\_items\_embeddings,neg\_items\_embeddings,parameters\_norm,  
 users\_embeddings\_layer0,pos\_items\_embeddings\_layer0,neg\_items\_embeddings\_layer0):  
 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]  
 # 计算正样本的得分，即用户嵌入向量和正样本嵌入向量的点积。  
 pos\_scores = torch.mul(users\_embeddings, pos\_items\_embeddings)  
 pos\_scores = torch.sum(pos\_scores, dim=1)  
 # 计算负样本的得分，即用户嵌入向量和负样本嵌入向量的点积。  
 neg\_scores = torch.mul(users\_embeddings, neg\_items\_embeddings)  
 neg\_scores = torch.sum(neg\_scores, dim=1)  
 # 计算损失。使用Softplus函数来转换负样本得分和正样本得分的差值，确保损失值始终为正。  
 loss = torch.mean(torch.nn.functional.softplus(neg\_scores - pos\_scores))  
 # 将正则化损失乘以权重衰减系数，并加到总损失上。  
 reg\_loss \*= self.weight\_decay  
 loss += reg\_loss  
 return loss

本次实验，选择使用**Adam优化算法**作为优化器，具体实现方式，如下所示。

optimizer = torch.optim.Adam(model.parameters(), lr=world.config["lr"])

1. **模型训练与测试**

**4.1 模型训练**

关于模型训练。首先，使用自定义的负采样方法，随机选择train\_data\_size个可重复的用户（train\_data\_size指训练数据中用户-项目的总交互数），依次分别随机采样一个对应的正例项目和一个负例项目，以获得train\_data\_size个训练样本[user, pos\_item, neg\_item]，将它们依次追加存储到训练列表samples中即可。然后，打乱训练集samlples中的样本顺序，在每个迭代周期中分批次计算BPRLoss并更新模型参数，并返回本次训练周期的平均损失和训练时间即可。

**4.2 测试评估**

每迭代训练10个周期，可以使用测试集对模型的推荐性能进行综合评估，并保存评估效果最好的模型参数。关于测试评估，具体如下所示。

1. 针对每个测试用户，排除该测试用户的正例项目，获得其前N个最高预测评分的项目。若N=2且测试用户列表users\_list=[user1, ...]，则推荐项目列表rating\_list=[[item3, item5], ...]，且对应的真实标签列表ground\_truth=[[user1’s all poc\_item in testDict], ...]。
2. 通过循环遍历不同top-N取值（1~N）的方式，考虑模型在不同top-N取值的情形下，测试用户在度量precision、recall、ndcg、diversity和novelty下的推荐性能。以recall为例，recall=[[TestUser1\_Recall at top-N=1, TestUser2\_Recall at top-N=1, ...], ...]。

（3）考虑到推荐的公平性问题，可按照历史交互项目数将不同量级的测试用户划分到不同分箱中，从而可以比较不同分箱测试用户组在不同top-N取值下的总体推荐性能。具体而言，需要循环遍历每个测试用户。首先，根据该用户的真实标签列表ground\_truth[user]和对应的预测标签列表label[user]（label=[[User\_PredItem1：isTrue 1 also 0, ...], ...]），计算该用户在不同Top-N取值情形下（若N=3，则只考虑label[user]中的前三个预测标签值）关于precision、recall及ndcg度量指标下的推荐性能。然后，按照每个用户所属的分箱索引和对应的top-N取值索引，将该用户的precision、recall及ndcg值分别累加存储到相应数组的索引位置处。至此，评估函数eval\_pairwise的返回结果，如下所示。

return {"recall": np.array(recall), "precision": np.array(precision), "ndcg": np.array(ndcg),  
 "diversity": utils.mean\_intra\_list\_distance(recom\_lists=sorted\_items, item\_embed=item\_embeddings),  
 "novelty": utils.novelty(ground\_truth, batch\_user\_interaction\_history, max(world.topks)),  
 "exploration\_vs\_precision": exploration\_vs\_precision,

"exploration\_vs\_recall": exploration\_vs\_recall,  
 "exploration\_vs\_ndcg": exploration\_vs\_ndcg

}

1. **实验细节**
2. **考虑归一化**

在图卷积神经网络中，考虑到节点度不同对卷积聚合的影响（模型可能会过度依赖热门项目或活跃用户）以及训练过程中的稳定性问题，可以对邻接矩阵进行归一化处理。同时，考虑到用户-项目交互矩阵只能记录一个方向上的交互，无法兼顾学习左侧节点（用户）对右侧节点（项目）的影响和右侧节点（项目）对左侧节点（用户）的影响，由此需要构造对称邻接矩阵来代替以完成后续工作，即：



关键在于，**如何将构造而来的对称邻接矩阵，进行归一化处理**。一般而言，可考虑通过计算度矩阵的逆进行L1归一化，也可考虑通过计算度矩阵的逆平方根进行L2归一化处理。更进一步，还可以考虑在不同的数据集中，从对称邻接矩阵的左侧（行）、右侧（列）和双边（先左侧，后右侧）方向分别进行L1或L2归一化处理，具体公式如下。



**实验结果**，如表2.1所示。其中，L2表示使用了L2归一化，L1表示使用了L1归一化，L表示左侧方向，R表示右侧方向，Both表示双边方向，LightGCN的默认参数为：decay=1e-4、lr=0.001、**layer=3**、seed=2020、topks=[20]、recdim=64、**epochs=100**。

表2.1 不同归一化方案对于LightGCN推荐性能的影响

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Gowalla**（稀疏性：8.4e-4） | | **A-Movies**（稀疏性：9.6e-4） | | **Yelp2018**（稀疏性：13e-4） | |
| **Recall@20** | **NDCG@20** | **Recall@20** | **NDCG@20** | **Recall@20** | **NDCG@20** |
| LGCN-L2-Both | 0.1406 | 0.1211 | 0.0930 | 0.0572 | 0.0469 | 0.0382 |
| LGCN-L2-L | 0.1262 | 0.1115 | 0.0834 | 0.0520 | 0.0400 | 0.0334 |
| LGCN-L2-R | 0.1433 | 0.1138 | 0.0791 | 0.0480 | 0.0493 | 0.0396 |
| LGCN-L1-Both | 0.1295 | 0.1127 | 0.0611 | 0.0396 | 0.0403 | 0.0330 |
| LGCN-L1-L | 0.1043 | 0.0906 | 0.0989 | 0.0636 | 0.0341 | 0.0277 |
| LGCN-L1-R | 0.1093 | 0.1445 | 0.0743 | 0.0453 | 0.0533 | 0.0415 |

表2.2 实验涉及数据集的具体情况

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset** | **Gowalla** | **A-Movies** | **Yelp2018** |
| User\_num | 29,858 | 44,438 | 31,668 |
| Item\_num | 40,981 | 25,046 | 38,048 |
| Inter\_num | 1,027,370 | 1,070,860 | 1,561,406 |
| Density | 8.4e-4 | 9.6e-4 | 13e-4 |

参考表2.2，可进行如下**实验结果分析**：

1. 从**数据稀疏性**的角度出发。在交互数Inter\_num量级相同的前提下，数据集稀疏程度和推荐性能呈正相关，即：LightGCN在更稀疏的数据上能取得更显著的性能表现。这或许是因为LightGCN仅需学习初始嵌入层的权重参数，以及通过求和平均不同层次嵌入结果来获得最终节点表示的特性，使其能够更简洁高效地利用稀疏数据中的信息并避免过拟合问题。然而，有限的可学习参数可能会难以有效捕捉稠密数据中蕴含的复杂关系，且难以应对稠密数据集上的过拟合问题。
2. 从**用户数和项目数比例**的角度出发。用户数更少的数据集Gowalla及Yelp2018，都在邻接矩阵右侧进行归一化的设置下（即：），取得了最佳性能表现。项目数更少的数据集A-Movies，则在对邻接矩阵的左侧进行归一化时（即：），取得了最佳性能表现。这是因为当用户数少于项目数时，通常项目有更多的连接，从而使得项目节点的度通常较高。由此，当使用度矩阵的逆对邻接矩阵右侧进行L1归一化时，可以使得度较低的用户节点在传播过程中的影响力被增强，从而有助于捕捉用户的独特偏好。而当用户数多于项目数时，也是同样的道理。
3. 从**L1和L2归一化**的角度出发。从不同方向对邻接矩阵进行归一化处理时，大多数情况下LightGCN在使用L2范数时的性能表现更好。相比之下，多个基于L1范数的归一化方案，在迭代训练中期就出现了显著的过拟合问题（表现出了：使用dropout等正则化手段，对于性能提升的潜力）。然而，对于异常值更敏感的L1范数而言，如若根据数据集中用户和项目数的比例关系，选择在特定的位置处进行L1归一化处理（比如，在Gowalla邻接矩阵的右侧进行L1归一化，其在NDCG上取得了优势明显的性能表现），反而能获得更激进的最佳性能表现。
4. **考虑注意力**

引入注意力修改LightGCN的**方案**有：引入可学习的注意力权重考虑如何组合各层生成的嵌入结果，用于代替原文直接将不同层的嵌入结果加和平均的嵌入组合方式；或进一步地引入解耦的用户和项目注意力考虑如何嵌入组合。也可以将包含各层嵌入结果的embs同时赋值给Q、K、V，从而通过计算缩放点积注意力的方式，代替原本包含各层嵌入结果的embs，继而完成后续的嵌入组合；或进一步将根据embs赋值而来的Q、K、V矩阵嵌入维，分别映射到特定维数的查询、键或值空间中去，以获得可调整Q、K、V矩阵嵌入维数的灵活性。

至此，针对前述四种LightGCN网络架构的修改方案，**实验结果**，如表2.3所示。其中，lgn表示基本LightGCN，base-a-lgn表示引入注意力考虑嵌入组合，finer-a-lgn表示引入解耦的用户和物品注意力考虑嵌入组合，sdp-a-lgn表示引入缩放点积注意力来代替包含各层嵌入结果的embs，w-sdp-a-lgn表示引入带有线性投影的缩放点积注意力来代替embs，实验默认参数为：decay=1e-4、lr=0.001、layer=3、seed=2024、topks=[20]、recdim=64、epochs=100、l1=False、side\_norm=both。

表2.3 不同注意力引入方案对于LightGCN推荐性能的影响

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Gowalla**（稀疏性：8.4e-4） | | | | | |
| **Recall@20** | **Precision@20** | **NDCG@20** | **Diversity** | | **备注** |
| lgn | 0.1393 | 0.0419 | 0.1198 | | 309.71 |  |
| base-a-lgn | 0.0790 | 0.0219 | 0.0578 | | 294.93 |  |
| finer-a-lgn | 0.0817 | 0.0226 | 0.0593 | | 291.16 |  |
| sdp-a-lgn | 0.1341 | 0.0396 | 0.1069 | | 527.19 |  |
| w-sdp-a-lgn | 0.0496 | 0.0129 | 0.0270 | | 505.55 | attention\_dim=8 |

参考表2.3，可进行如下**实验结果分析**：

1. 关于引入可学习的注意力权重**考虑嵌入组合**（base-a-lgn）。实验证明，额外使用随机初始化的线性层学习嵌入组合权重各分量的取值，实际效果不如原文将不同层次的嵌入结果直接加和平均。本质上，不同卷积聚合层生成的嵌入平滑结果，表达了不同层次的抽象关联（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）。因此，伴随图卷积层的增加，邻域聚合可能会导致嵌入表征的取值愈加平滑。至此，在迭代训练的过程中，由于可学习嵌入组合权重各分量取值的剧烈变化，可能严重影响了嵌入组合关于应对嵌入结果过平滑问题的实际效果。
2. 关于引入解耦的用户和物品注意力权重**考虑嵌入组合**（finer-a-lgn），本方案学习获得的每个维度为（user\_num+item\_num, 1）的嵌入组合权重分量，都对应一个卷积聚合层生成的嵌入结果，从而实现了针对每个用户和物品节点的精细化嵌入组合考量。实验证明，相较于base-a-lgn采用维度为（1, layer+1, 1）的可学习嵌入组合权重，这种细粒度更高的嵌入组合方法，能够一定程度上提高嵌入组合的效果。然而，该方案本质上仍和base-a-lgn方法一样，缺乏应对深层卷积聚合结果的过平滑问题。换而言之，这也印证了原文嵌入组合方式的简洁性（即：无可学习参数）与对嵌入结果过平滑问题的应对能力。
3. 关于引入缩放点积注意力**重构包含各层嵌入结果的embs**（sdp-a-lgn），继而用于后续的嵌入组合工作。本方案通过将注意力权重与值矩阵embs相乘的方式，企图捕捉每个节点在图中的重要性以及和其它节点间的关联程度。实验表明，本方案没有显著提升LightGCN模型的推荐性能，其仅在推荐结果的多样性方面提升明显。这可能是因为引入自注意力机制重构各层生成的嵌入结果，导致了节点嵌入的表示学习发生了变化，使得LightGCN更注重节点间的关联性，而非节点本身的属性特征。这将导致LightGCN更倾向于探索与用户历史行为相似的新项目，从而提升了推荐的多样性，但其对于推荐的召回率和精确度影响较小。
4. 关于引入带有线性投影的缩放点积注意力**重构embs**（w-sdp-a-lgn）。相较于sdp-a-lgn，本方案提供了手动灵活调整Q、K、V矩阵嵌入维数的灵活性。考虑到运行时间，本次实验并没有充分尝试超参attention\_dim不同取值，对于LightGCN模型推荐性能的影响。仅就attention\_dim=8时而言，其在召回率及NDCG等方面的推荐性能表现不佳。