代码运行说明

# 1.软件概述

## 1.1软件目标

使用MovieLens数据集，采用神经协同过滤（NCF）模型开发并实现一个推荐算法。

## 1.2软件功能

针对输入的用户id，推荐其最有可能喜欢的10部电影，打印电影名与预测打分值。

# 2.运行环境

操作系统名称及版本号：windows 10

GPU：NVIDIA GeForce GTX 1650 Ti

CUDA：11.8

编程语言：python3.9

实现平台：pycharm，anaconda3虚拟环境

重要依赖库：

pandas 1.5.3

pytorch 11.8

numpy 1.22.4

sklearn 0.0.post5

matplotlib 3.7.1

# 3.使用说明

## 3.1安装和初始化

下载程序后安装相应依赖包，配置解释器

## 3.2数据说明

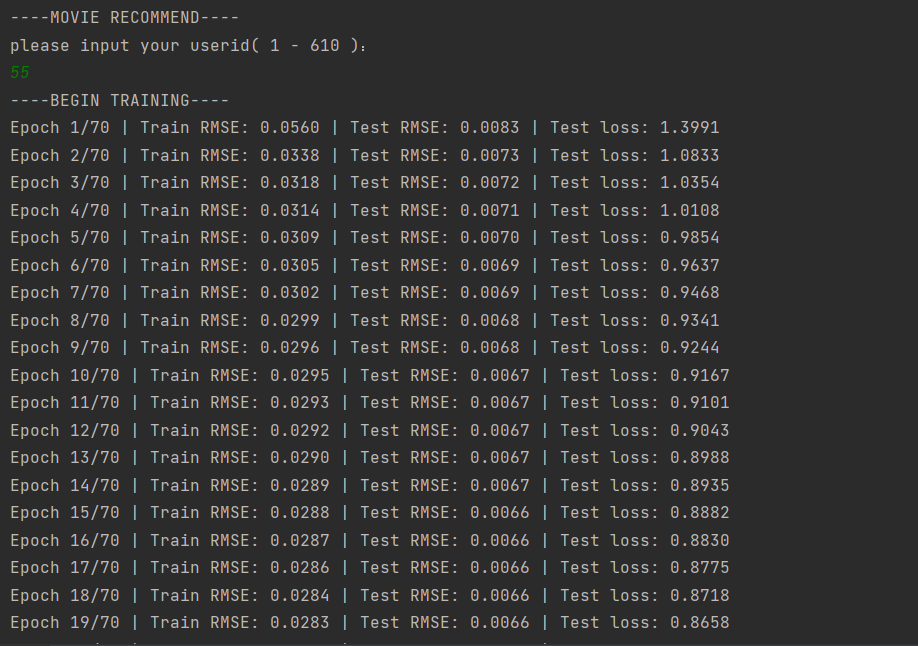
本软件使用MOVIELENS中ml-latest-small数据集，使用其中movies.csv,ratings.csv两份数据集。

格式如下：

movies.csv：movieId title genres共9742部电影

ratings.csv：userId movieId rating timestamp共610个用户的100836条评价

# 4.运行说明

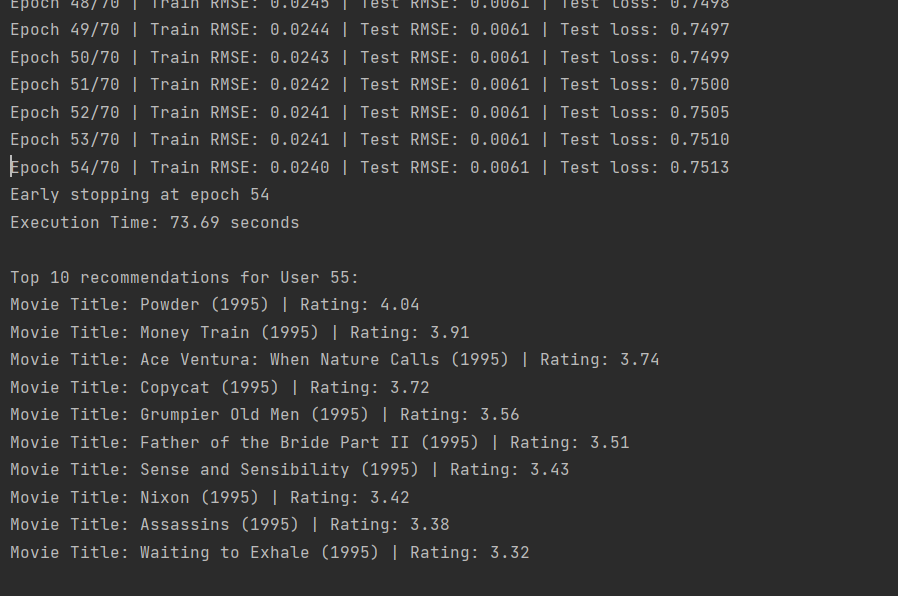


1.运行程序后，显示----MOVIE RECOMMEND----即本程序主要功能。

2.随后提示please input your userid( 1 - 610 )：要求用户从键盘输入自己的id，在1到610之间（与数据集一致），若输入不合法提示please input correct id。

3.用户输入正确后显示----BEGIN TRAINING----即开始训练。

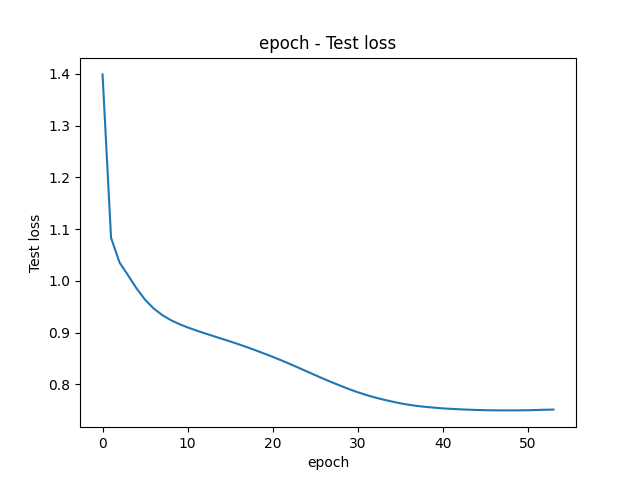
4.训练过程中没一个轮次打印：Epoch 1/70 | Train RMSE: 0.0546 | Test RMSE: 0.0083 | Test loss: 1.3863显示本次训练轮次，训练集与测试集均方根误差（用于评估模型性能）以及测试损失。



1.训练结束后打印早停信息：Early stopping at epoch 54即在54轮训练后提前停止。

2.打印总执行时间：Execution Time: 73.69 seconds。

3.随后打印本用户最可能喜欢的10部电影信息，包括名称和预测打分值。



程序同时输出上图，显示test loss和epoch训练轮次之间的关系，为训练过程的可视化。

# 5.总体思路

1. 首先检查硬件是否正常并加载数据
2. 将数据集进行分割，分为训练集和测试集
3. 设定各种参数
4. 对异常进行处理，并添加早停机制，提高效率
5. 构建模型
6. 形成模型实例，和数据一起移动到GPU上
7. 训练模型
8. 生成推荐
9. 打印指定用户的推荐电影

# 6.代码详细注释

## 检查硬件并加载数据集

# 测试训练时间

start\_time = time.time()

# 检查GPU可用性

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 加载电影元数据

movies\_metadata = pd.read\_csv('movies.csv', usecols=['movieId', 'title'])

movie\_id\_to\_title = dict(zip(movies\_metadata['movieId'], movies\_metadata['title']))

# 加载数据集

data = np.loadtxt('ratings.csv', delimiter=',', skiprows=1)

user\_ids = data[:, 0]

movie\_ids = data[:, 1]

ratings = data[:, 2]

## 分割数据集

# 数据集划分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

np.column\_stack((user\_ids, movie\_ids)), ratings, test\_size=0.2, random\_state=42

)

# 唯一化movie\_ids，并重新构建字典

unique\_movie\_ids = np.unique(movie\_ids)

movie\_id\_to\_title = dict(zip(movies\_metadata['movieId'], movies\_metadata['title']))

## 设定参数

# 超参数设置

num\_users = int(max(user\_ids)) + 1

num\_movies = int(max(movie\_ids)) + 1

embedding\_dim = 64 # 嵌入维度，用于表示用户和电影的向量表示的维度大小

hidden\_dim = 64 # 隐藏层维度，用于神经网络模型中隐藏层的大小

learning\_rate = 0.001 # 学习率，用于控制模型在训练过程中参数更新的速度

num\_epochs = 70 # 训练轮数，表示模型在整个训练集上的训练次数

batch\_size = 1024 # 批量大小，用于指定每个训练批次的样本数量

## 异常处理

print("----MOVIE RECOMMEND----")

print('please input your userid( 1 -',num\_users-1,')：')

# 异常处理：检查输入id合法性

def check():

while True:

try:

user\_id = int(input())

if 0 < user\_id < num\_users:

return user\_id

else:

print("please input correct id")

except ValueError as err:

print("please input correct id")

user\_id = check()

print("----BEGIN TRAINING----")

## 早停机制

#如果训练效果不理想，提前结束训练，提高效率

# 添加早停机制

best\_test\_rmse = np.inf

patience = 5

early\_stopping\_counter = 0

# 早停机制

if test\_rmse < best\_test\_rmse:

best\_test\_rmse = test\_rmse

early\_stopping\_counter = 0

else:

early\_stopping\_counter += 1

if early\_stopping\_counter >= patience:

print(f'Early stopping at epoch {epoch+1}')

break

## 构建模型

class Recommender(nn.Module):

def \_\_init\_\_(self, num\_users, num\_movies, embedding\_dim, hidden\_dim, lambda\_reg):

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

# 嵌入层构建

self.user\_embedding = nn.Embedding(num\_users, embedding\_dim)

self.movie\_embedding = nn.Embedding(num\_movies, embedding\_dim)

# 线性层构建

self.fc1 = nn.Linear(2 \* embedding\_dim, hidden\_dim)

self.fc2 = nn.Linear(hidden\_dim, 1)

#ReLU激活函数，增加了模型的非线性能力

self.relu = nn.ReLU()

self.lambda\_reg = lambda\_reg

# 前向传播

def forward(self, inputs):

user\_idx = inputs[:, 0]

movie\_idx = inputs[:, 1]

user\_embedded = self.user\_embedding(user\_idx)

movie\_embedded = self.movie\_embedding(movie\_idx)

x = torch.cat((user\_embedded, movie\_embedded), dim=1)

x = self.relu(self.fc1(x))

x = self.fc2(x)

x = torch.sigmoid(x) # 添加sigmoid函数

reg\_loss = self.lambda\_reg \* (torch.norm(self.user\_embedding.weight) + torch.norm(self.movie\_embedding.weight))

return x.view(-1) \* 5.0 + reg\_loss # 将输出缩放到0到5之间

## 转移模型实例和数据

# 正则化强度参数设置

lambda\_reg = 0.001

model = Recommender(num\_users, num\_movies, embedding\_dim, hidden\_dim, lambda\_reg).to(device)

# 定义损失函数和优化器

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate, weight\_decay=lambda\_reg)

# 将数据转换为PyTorch张量并移动到GPU上

X\_train = torch.LongTensor(X\_train).to(device)

y\_train = torch.FloatTensor(y\_train).clamp(0, 5).to(device) # 限制评分在0到5之间

X\_test = torch.LongTensor(X\_test).to(device)

y\_test = torch.FloatTensor(y\_test).clamp(0, 5).to(device) # 限制评分在0到5之间

## 训练模型与评估模型

# 训练模型

for epoch in range(num\_epochs):

model.train()

running\_loss = 0.0

for i in range(0, len(X\_train), batch\_size):

inputs = X\_train[i:i + batch\_size] # 从训练集中获取当前批次的输入

targets = y\_train[i:i + batch\_size] # 从训练集中获取当前批次的目标输出

optimizer.zero\_grad() # 将优化器的梯度缓冲区清零，以准备进行反向传播

outputs = model(inputs) # 通过向前传递输入数据，获取模型的输出

loss = criterion(outputs, targets) # 计算模型输出与目标输出之间的损失

running\_loss += loss.item()

loss.backward() # 执行反向传播，计算参数的梯度

optimizer.step() # 根据计算得到的梯度，更新模型的参数

train\_rmse = np.sqrt(running\_loss / len(X\_train)) # 计算训练集上的均方根误差（RMSE），用于评估模型在训练集上的性能

# 在测试集上进行评估

model.eval()

with torch.no\_grad():

outputs = model(X\_test) # 通过向前传递测试集的输入数据，获取模型在测试集上的输出

test\_loss = criterion(outputs, y\_test) # 计算模型在测试集上的输出与真实标签之间的损失

test\_rmse = np.sqrt(test\_loss.item() / len(X\_test))

linex.append(epoch)

liney.append(test\_loss.cpu())

print(f'Epoch {epoch + 1}/{num\_epochs} | Train RMSE: {train\_rmse:.4f} | Test RMSE: {test\_rmse:.4f} | Test loss: {test\_loss:.4f}') # 监控训练过程中模型的性能表现

## 训练可视化

# 画图

linex = []

liney = []

# 创建图形并绘制曲线

plt.plot(linex, liney)

# 添加标题和坐标轴标签

plt.title("epoch - Test loss")

plt.xlabel("epoch")

plt.ylabel("Test loss")

## 生成推荐

# 计算训练时长

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f'Execution Time: {execution\_time:.2f} seconds')

print()

# 生成推荐

user\_ids = torch.LongTensor(user\_ids).to(device)

movie\_ids = torch.LongTensor(movie\_ids).to(device)

recommendations = model(torch.column\_stack((user\_ids, movie\_ids))).detach().cpu().numpy()

## 对某个用户进行推荐

# 打印某个用户的前10个推荐电影

top\_k = 10

user\_recommendations = recommendations[user\_ids.cpu() == user\_id]

top\_movies = np.argsort(user\_recommendations)[-top\_k:][::-1]

print(f'Top {top\_k} recommendations for User {user\_id}:')

for movie\_id in top\_movies:

movie\_title = movie\_id\_to\_title.get(unique\_movie\_ids[movie\_id])

print(f'Movie Title: {movie\_title} | Rating: {user\_recommendations[movie\_id]:.2f}')

plt.show()