```python
#!/usr/bin/env python
# coding: utf-8

# 模型设计的代码需要用到上一节数据处理的Python类，定义如下：

# In[1]:


import random
import numpy as np
from PIL import Image

# 数据集总数据数： 1000209
# 单条数据集数据： {'usr_info': {'usr_id': 2, 'gender': 0, 'age': 56, 'job': 16}, 'mov_info': {'mov_id': 3654, 'title': [2337, 11, 4926, 26, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'category': [8, 7, 14, 0, 0, 0], 'years': 1961}, 'scores': 3.0}


class MovieLen(object):
    def __init__(self, use_poster):
        self.use_poster = use_poster
        # 声明每个数据文件的路径
        usr_info_path = "./work/ml-1m/users.dat"
        if use_poster:
            rating_path = "./work/ml-1m/new_rating.txt"
        else:
            rating_path = "./work/ml-1m/ratings.dat"

        movie_info_path = "./work/ml-1m/movies.dat"
        self.poster_path = "./work/ml-1m/posters/"
        # 得到电影数据
        self.movie_info, self.movie_cat, self.movie_title = self.get_movie_info(movie_info_path)
        # 记录电影的最大ID
        self.max_mov_cat = np.max([self.movie_cat[k] for k in self.movie_cat])
        self.max_mov_tit = np.max([self.movie_title[k] for k in self.movie_title])
        self.max_mov_id = np.max(list(map(int, self.movie_info.keys())))
        # 记录用户数据的最大ID
        self.max_usr_id = 0
        self.max_usr_age = 0
        self.max_usr_job = 0
        # 得到用户数据
        self.usr_info = self.get_usr_info(usr_info_path)
        # 得到评分数据
        self.rating_info = self.get_rating_info(rating_path)
        # 构建数据集
        self.dataset = self.get_dataset(usr_info=self.usr_info,
                                        rating_info=self.rating_info,
                                        movie_info=self.movie_info)
        # 划分数据集，获得数据加载器
        self.train_dataset = self.dataset[:int(len(self.dataset)*0.9)]
        self.valid_dataset = self.dataset[int(len(self.dataset)*0.9):]
        print("##Total dataset instances: ", len(self.dataset))
        print("##MovieLens dataset information: \nusr num: {}\n"
              "movies num: {}".format(len(self.usr_info),len(self.movie_info)))
    # 得到电影数据
    def get_movie_info(self, path):
        # 打开文件，编码方式选择ISO-8859-1，读取所有数据到data中
        with open(path, 'r', encoding="ISO-8859-1") as f:
            data = f.readlines()
        # 建立三个字典，分别用户存放电影所有信息，电影的名字信息、类别信息
        movie_info, movie_titles, movie_cat = {}, {}, {}
        # 对电影名字、类别中不同的单词计数
        t_count, c_count = 1, 1

        count_tit = {}
        # 按行读取数据并处理
        for item in data:
            item = item.strip().split("::")
            v_id = item[0]
            v_title = item[1][:-7]
            cats = item[2].split('|')
```

```
            v_year = item[1][-5:-1]

            titles = v_title.split()
            # 统计电影名字的单词，并给每个单词一个序号，放在movie_titles中
            for t in titles:
                if t not in movie_titles:
                    movie_titles[t] = t_count
                    t_count += 1
            # 统计电影类别单词，并给每个单词一个序号，放在movie_cat中
            for cat in cats:
                if cat not in movie_cat:
                    movie_cat[cat] = c_count
                    c_count += 1
            # 补0使电影名称对应的列表长度为15
            v_tit = [movie_titles[k] for k in titles]
            while len(v_tit)<15:
                v_tit.append(0)
            # 补0使电影种类对应的列表长度为6
            v_cat = [movie_cat[k] for k in cats]
            while len(v_cat)<6:
                v_cat.append(0)
            # 保存电影数据到movie_info中
            movie_info[v_id] = {'mov_id': int(v_id),
                                'title': v_tit,
                                'category': v_cat,
                                'years': int(v_year)}
        return movie_info, movie_cat, movie_titles

    def get_usr_info(self, path):
        # 性别转换函数，M-0，  F-1
        def gender2num(gender):
            return 1 if gender == 'F' else 0

        # 打开文件，读取所有行到data中
        with open(path, 'r') as f:
            data = f.readlines()
        # 建立用户信息的字典
        use_info = {}

        max_usr_id = 0
        #按行索引数据
        for item in data:
            # 去除每一行中和数据无关的部分
            item = item.strip().split("::")
            usr_id = item[0]
            # 将字符数据转成数字并保存在字典中
            use_info[usr_id] = {'usr_id': int(usr_id),
                                'gender': gender2num(item[1]),
                                'age': int(item[2]),
                                'job': int(item[3])}
            self.max_usr_id = max(self.max_usr_id, int(usr_id))
            self.max_usr_age = max(self.max_usr_age, int(item[2]))
            self.max_usr_job = max(self.max_usr_job, int(item[3]))
        return use_info
    # 得到评分数据
    def get_rating_info(self, path):
        # 读取文件里的数据
        with open(path, 'r') as f:
            data = f.readlines()
        # 将数据保存在字典中并返回
        rating_info = {}
        for item in data:
            item = item.strip().split("::")
            usr_id,movie_id,score = item[0],item[1],item[2]
            if usr_id not in rating_info.keys():
                rating_info[usr_id] = {movie_id:float(score)}
            else:
                rating_info[usr_id][movie_id] = float(score)
        return rating_info
    # 构建数据集
    def get_dataset(self, usr_info, rating_info, movie_info):
        trainset = []
```

```
142             for usr_id in rating_info.keys():
143                 usr_ratings = rating_info[usr_id]
144                 for movie_id in usr_ratings:
145                     trainset.append({'usr_info': usr_info[usr_id],
146                                      'mov_info': movie_info[movie_id],
147                                      'scores': usr_ratings[movie_id]})
148             return trainset
149 
150         def load_data(self, dataset=None, mode='train'):
151             use_poster = False
152 
153             # 定义数据迭代Batch大小
154             BATCHSIZE = 256
155 
156             data_length = len(dataset)
157             index_list = list(range(data_length))
158             # 定义数据迭代加载器
159             def data_generator():
160                 # 训练模式下，打乱训练数据
161                 if mode == 'train':
162                     random.shuffle(index_list)
163                 # 声明每个特征的列表
164                 usr_id_list,usr_gender_list,usr_age_list,usr_job_list = [], [], [], []
165                 mov_id_list,mov_tit_list,mov_cat_list,mov_poster_list = [], [], [], []
166                 score_list = []
167                 # 索引遍历输入数据集
168                 for idx, i in enumerate(index_list):
169                     # 获得特征数据保存到对应特征列表中
170                     usr_id_list.append(dataset[i]['usr_info']['usr_id'])
171                     usr_gender_list.append(dataset[i]['usr_info']['gender'])
172                     usr_age_list.append(dataset[i]['usr_info']['age'])
173                     usr_job_list.append(dataset[i]['usr_info']['job'])
174 
175                     mov_id_list.append(dataset[i]['mov_info']['mov_id'])
176                     mov_tit_list.append(dataset[i]['mov_info']['title'])
177                     mov_cat_list.append(dataset[i]['mov_info']['category'])
178                     mov_id = dataset[i]['mov_info']['mov_id']
179 
180                     if use_poster:
181                         # 不使用图像特征时，不读取图像数据，加快数据读取速度
182                         poster = Image.open(self.poster_path+'mov_id{}.jpg'.format(str(
                            mov_id[0])))
183                         poster = poster.resize([64, 64])
184                         if len(poster.size) <= 2:
185                             poster = poster.convert("RGB")
186 
187                         mov_poster_list.append(np.array(poster))
188 
189                     score_list.append(int(dataset[i]['scores']))
190                     # 如果读取的数据量达到当前的batch大小，就返回当前批次
191                     if len(usr_id_list)==BATCHSIZE:
192                         # 转换列表数据为数组形式，reshape到固定形状
193                         usr_id_arr = np.array(usr_id_list)
194                         usr_gender_arr = np.array(usr_gender_list)
195                         usr_age_arr = np.array(usr_age_list)
196                         usr_job_arr = np.array(usr_job_list)
197 
198                         mov_id_arr = np.array(mov_id_list)
199                         mov_cat_arr = np.reshape(np.array(mov_cat_list), [BATCHSIZE, 6]).
                            astype(np.int64)
200                         mov_tit_arr = np.reshape(np.array(mov_tit_list), [BATCHSIZE, 1,
                            15]).astype(np.int64)
201 
202                         if use_poster:
203                             mov_poster_arr = np.reshape(np.array(mov_poster_list)/127.5 -
                                1, [BATCHSIZE, 3, 64, 64]).astype(np.float32)
204                         else:
205                             mov_poster_arr = np.array([0.])
206 
207                         scores_arr = np.reshape(np.array(score_list), [-1, 1]).astype(np.
                            float32)
208 
```

```
                    # 放回当前批次数据
                    yield [usr_id_arr, usr_gender_arr, usr_age_arr, usr_job_arr],
                                                [mov_id_arr, mov_cat_arr, mov_tit_arr
                    , mov_poster_arr], scores_arr

                    # 清空数据
                    usr_id_list, usr_gender_list, usr_age_list, usr_job_list = [],
                    [], [], []
                    mov_id_list, mov_tit_list, mov_cat_list, score_list = [], [], [],
                     []
                    mov_poster_list = []
        return data_generator


# In[1]:


# 解压数据集
get_ipython().system('unzip -o -q -d ~/work/ ~/data/data19736/ml-1m.zip')


# # 模型设计介绍

#
神经网络模型设计是电影推荐任务中重要的一环。它的作用是提取图像、文本或者语音的特征，利用这些特征完成分类、检测、文本分析等任务。在电影推荐任务中，我们将设计一个神经网络模型，提取用户数据、电影数据的特征向量，然后计算这些向量的相似度，利用相似度的大小去完成推荐。
#
# 根据第一章中对建模思路的分析，神经网络模型的设计包含如下步骤：
# 1. 分别将用户、电影的多个特征数据转换成特征向量。
# 2. 对这些特征向量，使用全连接层或者卷积层进一步提取特征。
# 3. 将用户、电影多个数据的特征向量融合成一个向量表示，方便进行相似度计算。
# 4. 计算特征之间的相似度。
#
# 依据这个思路，我们设计一个简单的电影推荐神经网络模型：
#
# <center><img src="
https://ai-studio-static-online.cdn.bcebos.com/0186dbde202646a2863ebc70869400affa4509f526d34e7b9c77da6c4554474a" width="800" ></center>
#
# <center><br>图1：网络结构的设计 </br></center>
# <br></br>
#

# 该网络结构包含如下内容：
#
# 1. 提取用户特征和电影特征作为神经网络的输入，其中：
#    * 用户特征包含四个属性信息，分别是用户ID、性别、职业和年龄。
#    * 电影特征包含三个属性信息，分别是电影ID、电影类型和电影名称。
#
# 2.
提取用户特征。使用Embedding层将用户ID映射为向量表示，输入全连接层，并对其他三个属性也做类似的处理。然后将四个属性的特征分别全连接并相加。
#
# 3.
提取电影特征。将电影ID和电影类型映射为向量表示，输入全连接层，电影名字用文本卷积神经网络得到其定长向量表示。然后将三个属性的特征表示分别全连接并相加。
#
# 4.
得到用户和电影的向量表示后，计算二者的余弦相似度。最后，用该相似度和用户真实评分的均方差作为该回归模型的损失函数。
# ><font
size=2>衡量相似度的计算有多种方式，比如计算余弦相似度、皮尔森相关系数、Jaccard相似系数等等，或者通过计算欧几里得距离、曼哈顿距离、明可夫斯基距离等方式计算相似度。余弦相似度是一种简单好用的向量相似度计算方式，通过计算向量之间的夹角余弦值来评估他们的相似度，本节我们使用余弦相似度计算特征之间的相似度。</font>

# ### 为何如此设计网络呢？
#
# 网络的主体框架已经在第一章中做出了分析，但还有一些细节点没有确定。
#
```

```
# 1. 如何将"数字"转变成"向量"？
#
#   如NLP章节的介绍，使用词嵌入（Embedding）的方式可将数字转变成向量。
#
# 2.
如何合并多个向量的信息？例如：如何将用户四个特征（ID、性别、年龄、职业）的向量合并成一个向量？
#
#   最简单的方式是先将不同特征向量（ID 32维、性别 16维、年龄 16维、职业
16维）通过4个全连接层映射到4个等长的向量（200维度），再将4个等长的向量按位相加即可得到1个包含全部信息的向量。
#
#
电影类型的特征是将多个数字（代表出现的单词）转变成的多个向量（6个），可以通过相同的方式合并成1个向量。
#
# 3. 如何处理文本信息？
#
#
如NLP章节的介绍，使用卷积神经网络(CNN)和长短记忆神经网络（LSTM）处理文本信息会有较好的效果。因为电影标题是相对简单的短文本，所以我们使用卷积网络结构来处理电影标题。
#
# 4. 尺寸大小应该如何设计？
#
这涉及到信息熵的理念：越丰富的信息，维度越高。所以，信息量较少的原始特征可以用更短的向量表示，例如性别、年龄和职业这三个特征向量均设置成16维，而用户ID和电影ID这样较多信息量的特征设置成32维。综合了4个原始用户特征的向量和综合了3个电影特征的向量均设计成200维度，使得它们可以蕴含更丰富的信息。当然，尺寸大小并没有一贯的最优规律，需要我们根据问题的复杂程度，训练样本量，特征的信息量等多方面信息探索出最有效的设计。
#
# 第一章的设计思想结合上面几个细节方案，即可得出上图展示的网络结构。
#
#
接下来我们进入代码实现环节，首先看看如何将数据映射为向量。在自然语言处理中，我们常使用词嵌入（Embedding）的方式完成向量变换。

# # Embedding介绍
#
# Embedding是一个嵌入层，将输入的非负整数矩阵中的每个数值，转换为具有固定长度的向量。
#
#
在NLP任务中，一般把输入文本映射成向量表示，以便神经网络的处理。在数据处理章节，我们已经将用户和电影的特征用数字表示。嵌入层Embedding可以完成数字到向量的映射。
#
#
# 飞桨支持[Embedding API](
https://www.paddlepaddle.org.cn/documentation/docs/en/develop/api/paddle/nn/layer/common/Embedding_en.html)
，该接口根据输入从Embedding矩阵中查询对应Embedding信息，并会根据输入参数num_embeddings和embedding_dim自动构造一个二维Embedding矩阵。
#
# > *class* paddle.nn.Embedding *(num_embeddings, embedding_dim, padding_idx=None,
sparse=False, weight_attr=None, name=None)*
#
# 常用参数含义如下：
#
# * num_embeddings (int)：表示嵌入字典的大小。
# * embedding_dim ：表示每个嵌入向量的大小。
# * sparse
(bool)：是否使用稀疏更新，在词嵌入权重较大的情况下，使用稀疏更新能够获得更快的训练速度及更小的内存/显存占用。
# * weight_attr (ParamAttr)：指定嵌入向量的配置，包括初始化方法，具体用法请参见
ParamAttr ，一般无需设置，默认值为None。
#
#
# 我们需要特别注意，embedding函数在输入Tensor
shape的最后一维后面添加embedding_dim的维度，所以输出的维度数量会比输入多一个。以下面的代码为例，当输入的Tensor尺寸是[1]、embedding_dim是32时，输出Tensor的尺寸是[1,32]。

# In[3]:


```

```
import paddle
from paddle.nn import Linear, Embedding, Conv2D
import numpy as np
import paddle.nn.functional as F
import paddle.nn as nn

# 声明用户的最大ID，在此基础上加1（算上数字0）
USR_ID_NUM = 6040 + 1
# 声明Embedding 层，将ID映射为32长度的向量
usr_emb = Embedding(num_embeddings=USR_ID_NUM,
                    embedding_dim=32,
                    sparse=False)
# 声明输入数据，将其转成tensor
arr_1 = np.array([1], dtype="int64").reshape((-1))
print(arr_1)
arr_pd1 = paddle.to_tensor(arr_1)
print(arr_pd1)
# 计算结果
emb_res = usr_emb(arr_pd1)
# 打印结果
print("数字 1 的embedding结果是：  ", emb_res.numpy(), "\n形状是：", emb_res.shape)

output:
[1]
Tensor(shape=[1], dtype=int64, place=Place(gpu:0), stop_gradient=True,
       [1])
数字 1 的embedding结果是：   [[ 0.02368815  0.01219996 -0.00823128 -0.02978373
0.015901   -0.01567403
  -0.02949063  0.01960909  0.00287736  0.02580381 -0.01716401  0.02730818
  -0.00820427  0.01684101 -0.02887885  0.00482129  0.00490872  0.01330269
  -0.02448237 -0.00270003 -0.01551332 -0.0038403   0.01186426  0.00623586
   0.01695438 -0.02498322  0.02353216  0.02606978 -0.003106    0.00167086
  -0.00091827 -0.00629074]]
形状是：  [1, 32]

#
使用Embedding时，需要注意``num_embeddings``和``embedding_dim``这两个参数。``num_embeddings``表示词表大小；``embedding_dim``表示Embedding层维度。
#
#
使用的ml-1m数据集的用户ID最大为6040，考虑到0号ID的存在，因此这里我们需要将num_embeddings设置为6041（=6040+1）。embedding_dim表示将数据映射为embedding_dim维度的向量。这里将用户ID数据1转换成了维度为32的向量表示。32是设置的超参数，读者可以自行调整大小。

#
通过上面的代码，我们简单了解了Embedding的工作方式，但是Embedding层是如何将数字映射为高维度的向量的呢？
#
# 实际上，Embedding层和Conv2D,
Linear层一样，Embedding层也有可学习的权重，通过矩阵相乘的方法对输入数据进行映射。Embedding中将输入映射成向量的实际步骤是：
#
# 1. 将输入数据转换成one-hot格式的向量；
#
# 2. one-hot向量和Embedding层的权重进行矩阵相乘得到Embedding的结果。
#
#
下面展示了另一个使用Embedding函数的案例。该案例从0到9的10个ID数字中随机取出了3个，查看使用默认初始化方式的Embedding结果，再查看使用KaimingNormal（0均值的正态分布）初始化方式的Embedding结果。实际上，无论使用哪种参数初始化的方式，这些参数都是要在后续的训练过程中优化的，只是更符合任务场景的初始化方式可以使训练更快收敛，部分场景可以取得略好的模型精度。

# In[4]:


# 声明用户的最大ID，在此基础上加1（算上数字0）
USR_ID_NUM = 10
# 声明Embedding 层，将ID映射为16长度的向量
usr_emb = Embedding(num_embeddings=USR_ID_NUM,
                    embedding_dim=16,
                    sparse=False)
```

```
# 定义输入数据，输入数据为不超过10的整数，将其转成tensor
arr = np.random.randint(0, 10, (3)).reshape((-1)).astype('int64')
print("输入数据是：", arr)
arr_pd = paddle.to_tensor(arr)
emb_res = usr_emb(arr_pd)
print("默认权重初始化embedding层的映射结果是：", emb_res.numpy())

# 观察Embedding层的权重
emb_weights = usr_emb.state_dict()
print(emb_weights.keys())

print("\n查看embedding层的权重形状：", emb_weights['weight'].shape)

# 声明Embedding 层，将ID映射为16长度的向量，自定义权重初始化方式
# 定义KaimingNorma初始化方式
init = nn.initializer.KaimingNormal()
param_attr = paddle.ParamAttr(initializer=init)

usr_emb2 = Embedding(num_embeddings=USR_ID_NUM,
                     embedding_dim=16,
                     weight_attr=param_attr)
emb_res = usr_emb2(arr_pd)
print("\KaimingNormal初始化权重embedding层的映射结果是：", emb_res.numpy())


# 上面代码中，我们在[0,
10]范围内随机产生了3个整数，因此数据的最大值为整数9，最小为0。因此，输入数据映射为每个
one-hot向量的维度是10，定义Embedding权重的第一个维度USR_ID_NUM为10。
#
# 这里输入的数据shape是[3, 1]，Embedding层的权重形状则是[10,
16]，Embedding在计算时，首先将输入数据转换成one-hot向量，one-hot向量的长度和Embedding
层的输入参数size的第一个维度有关。比如这里我们设置的是10，所以输入数据将被转换成维度为
[3, 10]的one-hot向量，参数size决定了Embedding层的权重形状。最终维度为[3,
10]的one-hot向量与维度为[10, 16]Embedding权重相乘，得到最终维度为[3, 16]的映射向量。
#
#
我们也可以对Embeding层的权重进行初始化，如果不设置初始化方式，则采用默认的初始化方式。
#
#
神经网络处理文本数据时，需要用数字代替文本，Embedding层则是将输入数字数据映射成了高维
向量，然后就可以使用卷积、全连接、LSTM等网络层处理数据了，接下来我们开始设计用户和电影
数据的特征提取网络。
#

# 理解Embedding后，我们就可以开始构建提取用户特征的神经网络了。
#
# <center><img src="
https://ai-studio-static-online.cdn.bcebos.com/7398a606f0f0421f8057d10b24830948e17254d
e53df466684ede26ed0873892" width="500" ></center>
# <center><br>图2：提取用户特征网络示意 </br></center>
#
# 用户特征网络主要包括：
# 1. 将用户ID数据映射为向量表示，通过全连接层得到ID特征。
# 2. 将用户性别数据映射为向量表示，通过全连接层得到性别特征。
# 3. 将用户职业数据映射为向量表示，通过全连接层得到职业特征。
# 4. 将用户年龄数据影射为向量表示，通过全连接层得到年龄特征。
# 5. 融合ID、性别、职业、年龄特征，得到用户的特征表示。
#
#
在用户特征计算网络中，我们对每个用户数据做embedding处理，然后经过一个全连接层，激活函
数使用ReLU，得到用户所有特征后，将特征整合，经过一个全连接层得到最终的用户数据特征，该
特征的维度是200维，用于和电影特征计算相似度。

# ## 1. 提取用户ID特征
#
#
开始构建用户ID的特征提取网络，ID特征提取包括两个部分。首先，使用Embedding将用户ID映射
为向量；然后，使用一层全连接层和ReLU激活函数进一步提取用户ID特征。
#
相比较电影类别和电影名称，用户ID只包含一个数字，数据更为简单。这里需要考虑将用户ID映射
为多少维度的向量合适，使用维度过大的向量表示用户ID容易造成信息冗余，维度过低又不足以表
示该用户的特征。理论上来说，如果使用二进制表示用户ID，用户最大ID是6040，小于2的13次方
```

```
，因此，理论上使用13维度的向量已经足够了，为了让不同ID的向量更具区分性，我们选择将用户ID映射为维度为32维的向量。
#
#
# 下面是用户ID特征提取代码实现：
#

# In[5]:


# 自定义一个用户ID数据
usr_id_data = np.random.randint(0, 6040, (2)).reshape((-1)).astype('int64')
print("输入的用户ID是:", usr_id_data)

USR_ID_NUM = 6040 + 1
# 定义用户ID的embedding层和fc层
usr_emb = Embedding(num_embeddings=USR_ID_NUM,
                    embedding_dim=32,
                    sparse=False)
usr_fc = Linear(in_features=32, out_features=32)

usr_id_var = paddle.to_tensor(usr_id_data)
usr_id_feat = usr_fc(usr_emb(usr_id_var))

usr_id_feat = F.relu(usr_id_feat)
print("用户ID的特征是： ", usr_id_feat.numpy(), "\n其形状是： ", usr_id_feat.shape)


#
注意到，将用户ID映射为one-hot向量时，Embedding层参数size的第一个参数是，在用户的最大ID基础上加上1。原因很简单，从上一节数据处理已经发现，用户ID是从1开始计数的，最大的用户ID是6040。并且已经知道通过Embedding映射输入数据时，是先把输入数据转换成one-hot向量。向量中只有一个 1 的向量才被称为one-hot向量，比如，0 用四维的on-hot向量表示是[1, 0 ,0 ,0]，同时，4维的one-hot向量最大只能表示3。所以，要把数字6040用one-hot向量表示，至少需要用6041维度的向量。
#
#
#
接下来我们会看到，类似的Embeding层也适用于处理用户性别、年龄和职业，以及电影ID等特征，实现代码均是类似的。

# ## 2. 提取用户性别特征
#
#
接下来构建用户性别的特征提取网络，同用户ID特征提取步骤，使用Embedding层和全连接层提取用户性别特征。用户性别不像用户ID数据那样有数千数万种不同数据，性别只有两种可能，不需要使用高维度的向量表示其特征，这里我们将用户性别用为16维的向量表示。
#
# 下面是用户性别特征提取实现：

# In[6]:


# 自定义一个用户性别数据
usr_gender_data = np.array((0, 1)).reshape(-1).astype('int64')
print("输入的用户性别是:", usr_gender_data)

# 用户的性别用0， 1 表示
# 性别最大ID是1，所以Embedding层size的第一个参数设置为1 + 1 = 2
USR_ID_NUM = 2
# 对用户性别信息做映射，并紧接着一个FC层
USR_GENDER_DICT_SIZE = 2
usr_gender_emb = Embedding(num_embeddings=USR_GENDER_DICT_SIZE,
                           embedding_dim=16)

usr_gender_fc = Linear(in_features=16, out_features=16)

usr_gender_var = paddle.to_tensor(usr_gender_data)
usr_gender_feat = usr_gender_fc(usr_gender_emb(usr_gender_var))
usr_gender_feat = F.relu(usr_gender_feat)
print("用户性别特征的数据特征是： ", usr_gender_feat.numpy(), "\n其形状是： ",
usr_gender_feat.shape)
```

```
print("\n性别 0 对应的特征是：", usr_gender_feat.numpy()[0, :])
print("性别 1 对应的特征是：", usr_gender_feat.numpy()[1, :])


# ## 3. 提取用户年龄特征
# 然后构建用户年龄的特征提取网络，同样采用Embedding层和全连接层的方式提取特征。
#
# 前面我们了解到年龄数据分布是：
# * 1: "Under 18"
# * 18: "18-24"
# * 25: "25-34"
# * 35: "35-44"
# * 45: "45-49"
# * 50: "50-55"
# * 56: "56+"
#
# 得知用户年龄最大值为56，这里仍将用户年龄用16维的向量表示。

# In[7]:


# 自定义一个用户年龄数据
usr_age_data = np.array((1, 18)).reshape(-1).astype('int64')
print("输入的用户年龄是:", usr_age_data)

# 对用户年龄信息做映射，并紧接着一个Linear层
# 年龄的最大ID是56，所以Embedding层size的第一个参数设置为56 + 1 = 57
USR_AGE_DICT_SIZE = 56 + 1

usr_age_emb = Embedding(num_embeddings=USR_AGE_DICT_SIZE,
                        embedding_dim=16)
usr_age_fc = Linear(in_features=16, out_features=16)

usr_age = paddle.to_tensor(usr_age_data)
usr_age_feat = usr_age_emb(usr_age)
usr_age_feat = usr_age_fc(usr_age_feat)
usr_age_feat = F.relu(usr_age_feat)

print("用户年龄特征的数据特征是：", usr_age_feat.numpy(), "\n其形状是：",
usr_age_feat.shape)
print("\n年龄 1 对应的特征是：", usr_age_feat.numpy()[0, :])
print("年龄 18 对应的特征是：", usr_age_feat.numpy()[1, :])


# ## 4. 提取用户职业特征
#
#
参考用户年龄的处理方式实现用户职业的特征提取，同样采用Embedding层和全连接层的方式提取特征。由上一节信息可以得知用户职业的最大数字表示是20。

# In[8]:


# 自定义一个用户职业数据
usr_job_data = np.array((0, 20)).reshape(-1).astype('int64')
print("输入的用户职业是:", usr_job_data)

# 对用户职业信息做映射，并紧接着一个Linear层
# 用户职业的最大ID是20，所以Embedding层size的第一个参数设置为20 + 1 = 21
USR_JOB_DICT_SIZE = 20 + 1
usr_job_emb = Embedding(num_embeddings=USR_JOB_DICT_SIZE,embedding_dim=16)
usr_job_fc = Linear(in_features=16, out_features=16)

usr_job = paddle.to_tensor(usr_job_data)
usr_job_feat = usr_job_emb(usr_job)
usr_job_feat = usr_job_fc(usr_job_feat)
usr_job_feat = F.relu(usr_job_feat)

print("用户年龄特征的数据特征是：", usr_job_feat.numpy(), "\n其形状是：",
usr_job_feat.shape)
print("\n职业 0 对应的特征是：", usr_job_feat.numpy()[0, :])
print("职业 20 对应的特征是：", usr_job_feat.numpy()[1, :])
```

```
541
542
543 # ## 5. 融合用户特征
544 #
545 #
    特征融合是一种常用的特征增强手段，通过结合不同特征的长处，达到取长补短的目的。简单的融合方法有：特征（加权）相加、特征级联、特征正交等等。此处使用特征融合是为了将用户的多个特征融合到一起，用单个向量表示每个用户，更方便计算用户与电影的相似度。上文使用Embedding加全连接的方法，分别得到了用户ID、年龄、性别、职业的特征向量，可以使用全连接层将每个特征映射到固定长度，然后进行相加，得到融合特征。
546
547 # In[9]:
548
549
550 FC_ID = Linear(in_features=32, out_features=200)
551 FC_JOB = Linear(in_features=16, out_features=200)
552 FC_AGE = Linear(in_features=16, out_features=200)
553 FC_GENDER = Linear(in_features=16, out_features=200)
554
555 # 收集所有的用户特征
556 _features = [usr_id_feat, usr_job_feat, usr_age_feat, usr_gender_feat]
557 _features = [k.numpy() for k in _features]
558 _features = [paddle.to_tensor(k) for k in _features]
559
560 id_feat = F.tanh(FC_ID(_features[0]))
561 job_feat = F.tanh(FC_JOB(_features[1]))
562 age_feat = F.tanh(FC_AGE(_features[2]))
563 genger_feat = F.tanh(FC_GENDER(_features[-1]))
564
565 # 对特征求和
566 usr_feat = id_feat + job_feat + age_feat + genger_feat
567 print("用户融合后特征的维度是：", usr_feat.shape)
568
569
570 # 这里使用全连接层进一步提取特征，而不是直接相加得到用户特征的原因有两点：
571 # * 一是用户每个特征数据维度不一致，无法直接相加；
572 # *
    二是用户每个特征仅使用了一层全连接层，提取特征不充分，多使用一层全连接层能进一步提取特征。而且，这里用高维度（200维）的向量表示用户特征，能包含更多的信息，每个用户特征之间的区分也更明显。
573 #
574 #
    上述实现中需要对每个特征都使用一个全连接层，实现较为复杂，一种简单的替换方式是，先将每个用户特征沿着长度维度进行级联，然后使用一个全连接层获得整个用户特征向量，两种方式的对比见下图：
575 #
576 # <center><img src="
    https://ai-studio-static-online.cdn.bcebos.com/88cd1178bf98472faeea3e157e49cc6eed3caf4c6e5644c5a4196a63908bc667" width="800" ></center>
577 # <center> 图3：两种特征方式对比示意 </center>
578 # <br>
579 #
580 # 两种方式均可实现向量的合并，虽然两者的数学公式不同，但它们的表达方式是类似的。
581 #
582 #
583 # 下面是方式2的代码实现。
584
585 # In[10]:
586
587
588 usr_combined = Linear(in_features=80, out_features=200)
589
590 # 收集所有的用户特征
591 _features = [usr_id_feat, usr_job_feat, usr_age_feat, usr_gender_feat]
592
593 print("打印每个特征的维度：", [f.shape for f in _features])
594
595 _features = [k.numpy() for k in _features]
596 _features = [paddle.to_tensor(k) for k in _features]
597
598 # 对特征沿着最后一个维度级联
599 usr_feat = paddle.concat(_features, axis=1)
```

```
usr_feat = F.tanh(usr_combined(usr_feat))
print("用户融合后特征的维度是：", usr_feat.shape)


# 上述代码中，使用了[paddle.concat API](http://www.paddlepaddle.org.cn/documentation/docs/zh/2.0-rc/api/paddle/tensor/manipulation/concat_cn.html#concat)，表示沿着第几个维度将输入数据级联到一起。
#
# > paddle.concat *(x, axis=0, name=None)*
#
# 常用参数含义如下：
#
# * x (list|tuple)：待联结的Tensor list或者Tensor tuple，x中所有Tensor的数据类型应该一致。
# * axis (int|Tensor，可选)：指定对输入x进行运算的轴，默认值为0。
#
#
#
至此我们已经完成了用户特征提取网络的设计，包括ID特征提取、性别特征提取、年龄特征提取、职业特征提取和特征融合模块，下面我们将所有的模块整合到一起，放到Python类中，完整代码实现如下：

# In[11]:


import random
import math
class Model(nn.Layer):
    def __init__(self, use_poster, use_mov_title, use_mov_cat, use_age_job,fc_sizes):
        super(Model, self).__init__()

        # 将传入的name信息和bool型参数添加到模型类中
        self.use_mov_poster = use_poster
        self.use_mov_title = use_mov_title
        self.use_usr_age_job = use_age_job
        self.use_mov_cat = use_mov_cat
        self.fc_sizes = fc_sizes

        # 使用上节定义的数据处理类，获取数据集的信息，并构建训练和验证集的数据迭代器
        Dataset = MovieLen(self.use_mov_poster)
        self.Dataset = Dataset
        self.trainset = self.Dataset.train_dataset
        self.valset = self.Dataset.valid_dataset
        self.train_loader = self.Dataset.load_data(dataset=self.trainset, mode='train')
        self.valid_loader = self.Dataset.load_data(dataset=self.valset, mode='valid')

        """ define network layer for embedding usr info """
        USR_ID_NUM = Dataset.max_usr_id + 1
        # 对用户ID做映射，并紧接着一个FC层
        self.usr_emb = Embedding(num_embeddings=USR_ID_NUM,embedding_dim=32)
        self.usr_fc = Linear(32, 32)

        # 对用户性别信息做映射，并紧接着一个FC层
        USR_GENDER_DICT_SIZE = 2
        self.usr_gender_emb = Embedding(num_embeddings=USR_GENDER_DICT_SIZE, embedding_dim=16)
        self.usr_gender_fc = Linear(16, 16)

        # 对用户年龄信息做映射，并紧接着一个FC层
        USR_AGE_DICT_SIZE = Dataset.max_usr_age + 1
        self.usr_age_emb = Embedding(num_embeddings=USR_AGE_DICT_SIZE,embedding_dim=16)
        self.usr_age_fc = Linear(16, 16)

        # 对用户职业信息做映射，并紧接着一个FC层
        USR_JOB_DICT_SIZE = Dataset.max_usr_job + 1
        self.usr_job_emb = Embedding(num_embeddings=USR_JOB_DICT_SIZE,embedding_dim=16)
        self.usr_job_fc = Linear(16, 16)

        # 新建一个FC层，用于整合用户数据信息
```

```
662             self.usr_combined = Linear(80, 200)
663 
664             # 新建一个Linear层，用于整合电影特征
665             self.mov_concat_embed = Linear(in_features=96, out_features=200)
666 
667             user_sizes = [200] + self.fc_sizes
668             acts = ["relu" for _ in range(len(self.fc_sizes))]
669             self._user_layers = []
670             for i in range(len(self.fc_sizes)):
671                 linear = Linear(
672                     in_features=user_sizes[i],
673                     out_features=user_sizes[i + 1],
674                     weight_attr=paddle.ParamAttr(
675                         initializer=nn.initializer.Normal(
676                             std=1.0 / math.sqrt(user_sizes[i]))))
677                 # 向模型中添加了一个 paddle.nn.Linear 子层
678                 self.add_sublayer('linear_user_%d' % i, linear)
679                 self._user_layers.append(linear)
680                 if acts[i] == 'relu':
681                     act = nn.ReLU()
682                     # 向模型中添加了一个 paddle.nn.ReLU() 子层
683                     self.add_sublayer('user_act_%d' % i, act)
684                     self._user_layers.append(act)
685 
686 
687         # 定义计算用户特征的前向运算过程
688         def get_usr_feat(self, usr_var):
689             """ get usr features"""
690             # 获取到用户数据
691             usr_id, usr_gender, usr_age, usr_job = usr_var
692             # 将用户的ID数据经过embedding和FC计算，得到的特征保存在feats_collect中
693             feats_collect = []
694             usr_id = self.usr_emb(usr_id)
695             usr_id = self.usr_fc(usr_id)
696             usr_id = F.relu(usr_id)
697             feats_collect.append(usr_id)
698 
699             # 计算用户的性别特征，并保存在feats_collect中
700             usr_gender = self.usr_gender_emb(usr_gender)
701             usr_gender = self.usr_gender_fc(usr_gender)
702             usr_gender = F.relu(usr_gender)
703 
704             feats_collect.append(usr_gender)
705             # 选择是否使用用户的年龄-职业特征
706             if self.use_usr_age_job:
707                 # 计算用户的年龄特征，并保存在feats_collect中
708                 usr_age = self.usr_age_emb(usr_age)
709                 usr_age = self.usr_age_fc(usr_age)
710                 usr_age = F.relu(usr_age)
711                 feats_collect.append(usr_age)
712                 # 计算用户的职业特征，并保存在feats_collect中
713                 usr_job = self.usr_job_emb(usr_job)
714                 usr_job = self.usr_job_fc(usr_job)
715                 usr_job = F.relu(usr_job)
716                 feats_collect.append(usr_job)
717 
718             # 将用户的特征级联，并通过FC层得到最终的用户特征
719             print([f.shape for f in feats_collect])
720             usr_feat = paddle.concat(feats_collect, axis=1)
721             user_features = F.tanh(self.usr_combined(usr_feat))
722             #通过3层全链接层，获得用于计算相似度的用户特征和电影特征
723             for n_layer in self._user_layers:
724                 user_features = n_layer(user_features)
725             return user_features
726 
727     #下面使用定义好的数据读取器，实现从用户数据读取到用户特征计算的流程：
728     ## 测试用户特征提取网络
729     fc_sizes=[128, 64, 32]
730     model = Model(use_poster=False, use_mov_title=True, use_mov_cat=True, use_age_job=
        True,fc_sizes=fc_sizes)
731     model.eval()
732 
```

```
data_loader = model.train_loader

for idx, data in enumerate(data_loader()):
    # 获得数据，并转为动态图格式，
    usr, mov, score = data
#         print(usr.shape)
    # 只使用每个Batch的第一条数据
    usr_v = [[var[0]] for var in usr]


    print("输入的用户ID数据：{}\n性别数据：{} \n年龄数据：{} \n职业数据{}".format(*
    usr_v))

    usr_v = [paddle.to_tensor(np.array(var)) for var in usr_v]
    usr_feat = model.get_usr_feat(usr_v)
    print("计算得到的用户特征维度是：", usr_feat.shape)
    break



#
上面使用了向量级联+全连接的方式实现了四个用户特征向量的合并，为了捕获特征向量的深层次语义信息，合并后的向量还加入了3层全链接结构。在下面处理电影特征的部分我们会看到使用另外一种向量合并的方式（向量相加）处理电影类型的特征(6个向量合并成1个向量)，然后再加上全连接。

#
# # 电影特征提取网络
#
#
接下来我们构建提取电影特征的神经网络，与用户特征网络结构不同的是，电影的名称和类别均有多个数字信息，我们构建网络时，对这两类特征的处理方式也不同。
#
# <center><img src="
https://ai-studio-static-online.cdn.bcebos.com/763a59c9ca304da4941e673cb13d7b75dba320ee9a8c4fa3981f03b53b2ae0ea"
# width="450" ></center>
#
#
# 电影特征网络主要包括：
# 1. 将电影ID数据映射为向量表示，通过全连接层得到ID特征。
# 2. 将电影类别数据映射为向量表示，对电影类别的向量求和得到类别特征。
# 3. 将电影名称数据映射为向量表示，通过卷积层计算得到名称特征。
#

# ## 1. 提取电影ID特征
#
与计算用户ID特征的方式类似，我们通过如下方式实现电影ID特性提取。根据上一节信息得知电影ID的最大值是3952。
#

# In[12]:


# 自定义一个电影ID数据
mov_id_data = np.array((1, 2)).reshape(-1).astype('int64')
# 对电影ID信息做映射，并紧接着一个FC层
MOV_DICT_SIZE = 3952 + 1
mov_emb = Embedding(num_embeddings=MOV_DICT_SIZE, embedding_dim=32)
mov_fc = Linear(32, 32)


print("输入的电影ID是:", mov_id_data)
mov_id_data = paddle.to_tensor(mov_id_data)
mov_id_feat = mov_fc(mov_emb(mov_id_data))
mov_id_feat = F.relu(mov_id_feat)
print("计算的电影ID的特征是", mov_id_feat.numpy(), "\n其形状是：", mov_id_feat.shape)
print("\n电影ID为 {} 计算得到的特征是：{}".format(mov_id_data.numpy()[0], mov_id_feat
.numpy()[0]))
print("电影ID为 {} 计算得到的特征是：{}".format(mov_id_data.numpy()[1], mov_id_feat.
numpy()[1]))

```

```
792
793 # ## 2. 提取电影类别特征
794 #
795 #
    与电影ID数据不同的是，每个电影有多个类别，提取类别特征时，如果对每个类别数据都使用一个全连接层，电影最多的类别数是6，会导致类别特征提取网络参数过多而不利于学习。我们对于电影类别特征提取的处理方式是：
796 # 1. 通过Embedding网络层将电影类别数字映射为特征向量；
797 # 2. 对Embedding后的向量沿着类别数量维度进行求和，得到一个类别映射向量；
798 # 3. 通过一个全连接层计算类别特征向量。
799 #
800 #
    数据处理章节已经介绍到，每个电影的类别数量是不固定的，且一个电影最大的类别数量是6，类别数量不足6的通过补0到6维。因此，每个类别的数据维度是6，每个电影类别有6个Embedding向量。我们希望用一个向量就可以表示电影类别，可以对电影类别数量维度降维，
801 # 这里对6个Embedding向量通过求和的方式降维，得到电影类别的向量表示。
802 #
803 # 下面是电影类别特征提取的实现方法：
804
805 # In[13]:
806
807
808 # 自定义一个电影类别数据
809 mov_cat_data = np.array(((1, 2, 3, 0, 0, 0), (2, 3, 4, 0, 0, 0))).reshape(2, -1).
    astype('int64')
810 # 对电影ID信息做映射，并紧接着一个Linear层
811 MOV_DICT_SIZE = 6 + 1
812 mov_emb = Embedding(num_embeddings=MOV_DICT_SIZE, embedding_dim=32)
813 mov_fc = Linear(in_features=32, out_features=32)
814
815 print("输入的电影类别是:", mov_cat_data[:, :])
816 mov_cat_data = paddle.to_tensor(mov_cat_data)
817 # 1. 通过Embedding映射电影类别数据；
818 mov_cat_feat = mov_emb(mov_cat_data)
819 # 2. 对Embedding后的向量沿着类别数量维度进行求和，得到一个类别映射向量；
820 mov_cat_feat = paddle.sum(mov_cat_feat, axis=1, keepdim=False)
821
822 # 3. 通过一个全连接层计算类别特征向量。
823 mov_cat_feat = mov_fc(mov_cat_feat)
824 mov_cat_feat = F.relu(mov_cat_feat)
825 print("计算的电影类别的特征是", mov_cat_feat.numpy(), "\n其形状是: ", mov_cat_feat.
    shape)
826 print("\n电影类别为 {} 计算得到的特征是: {}".format(mov_cat_data.numpy()[0, :],
    mov_cat_feat.numpy()[0]))
827 print("\n电影类别为 {} 计算得到的特征是: {}".format(mov_cat_data.numpy()[1, :],
    mov_cat_feat.numpy()[1]))
828
829
830 #
    待合并的6个向量具有相同的维度，直接按位相加即可得到综合的向量表示。当然，我们也可以采用向量级联的方式，将6个32维的向量级联成192维的向量，再通过全连接层压缩成32维度，代码实现上要臃肿一些。
831
832 # ## 3. 提取电影名称特征
833 #
834 # 与电影类别数据一样，每个电影名称具有多个单词。我们对于电影名称特征提取的处理方式是：
835 #
836 # 1. 通过Embedding映射电影名称数据，得到对应的特征向量；
837 # 2. 对Embedding后的向量使用卷积层+全连接层进一步提取特征；
838 # 3. 对特征进行降采样，降低数据维度。
839 #
840 #
    提取电影名称特征时，使用了卷积层加全连接层的方式提取特征。这是因为电影名称单词较多，最大单词数量是15，如果采用和电影类别同样的处理方式，即沿着数量维度求和，显然会损失很多信息。考虑到15这个维度较高，可以使用卷积层进一步提取特征，同时通过控制卷积层的步长，降低电影名称特征的维度。
841 #
842 #
    如果只是简单的经过一层或二层卷积后，特征的维度依然很大，为了得到更低维度的特征向量，有两种方式，一种是利用求和降采样的方式，另一种是继续使用神经网络层进行特征提取并逐渐降低特征维度。这里，我们采用"简单求和"的降采样方式压缩电影名称特征的维度，通过飞桨的[reduce_sum](
```

https://www.paddlepaddle.org.cn/documentation/docs/zh/2.0-beta/api/paddle/fluid/layers/reduce_sum_cn.html) API实现。

```
#
# 下面是提取电影名称特征的代码实现：
#


# In[14]:


# 自定义两个电影名称数据
mov_title_data = np.array(((1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
                           (2, 3, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))).reshape(2, 1, 15).astype('int64')
# 对电影名称做映射，紧接着FC和pool层
MOV_TITLE_DICT_SIZE = 1000 + 1
mov_title_emb = Embedding(num_embeddings=MOV_TITLE_DICT_SIZE, embedding_dim=32)
mov_title_conv = Conv2D(in_channels=1, out_channels=1, kernel_size=(3, 1), stride=(2, 1), padding=0)
# 使用 3 * 3卷积层代替全连接层
mov_title_conv2 = Conv2D(in_channels=1, out_channels=1, kernel_size=(3, 1), stride=1, padding=0)

mov_title_data = paddle.to_tensor(mov_title_data)
print("电影名称数据的输入形状：", mov_title_data.shape)
# 1. 通过Embedding映射电影名称数据；
mov_title_feat = mov_title_emb(mov_title_data)
print("输入通过Embedding层的输出形状：", mov_title_feat.shape)
# 2. 对Embedding后的向量使用卷积层进一步提取特征；
mov_title_feat = F.relu(mov_title_conv(mov_title_feat))
print("第一次卷积之后的特征输出形状：", mov_title_feat.shape)
mov_title_feat = F.relu(mov_title_conv2(mov_title_feat))
print("第二次卷积之后的特征输出形状：", mov_title_feat.shape)

batch_size = mov_title_data.shape[0]
# 3.
最后对特征进行降采样，keepdim=False会让输出的维度减少，而不是用[2,1,1,32]的形式占位；
mov_title_feat = paddle.sum(mov_title_feat, axis=2, keepdim=False)
print("reduce_sum降采样后的特征输出形状：", mov_title_feat.shape)

mov_title_feat = F.relu(mov_title_feat)
mov_title_feat = paddle.reshape(mov_title_feat, [batch_size, -1])
print("电影名称特征的最终特征输出形状：", mov_title_feat.shape)

print("\n计算的电影名称的特征是", mov_title_feat.numpy(), "\n其形状是：",
mov_title_feat.shape)
print("\n电影名称为 {} 计算得到的特征是：{}".format(mov_title_data.numpy()[0,:, 0],
mov_title_feat.numpy()[0]))
print("\n电影名称为 {} 计算得到的特征是：{}".format(mov_title_data.numpy()[1,:, 0],
mov_title_feat.numpy()[1]))


# 上述代码中，通过Embedding层已经获得了维度是[batch_size， 1， 15，
32]电影名称特征向量，因此，该特征可以视为是通道数量为1的特征图，很适合使用卷积层进一步提取特征。这里我们使用两个$3\times1$大小的卷积核的卷积层提取特征，输出通道保持不变，仍然是1。特征维度中15是电影名称中单词的数量（最大数量），使用$3\times1$的卷积核，由于卷积感受野的原因，进行卷积时会综合多个单词的特征，同时设置卷积的步长参数stride为(2, 1)，即可对电影名称的维度降维，同时保持每个名称的向量长度不变，以防过度压缩每个名称特征的信息。
#
#
从输出结果来看，第一个卷积层之后的输出特征维度依然较大，可以使用第二个卷积层进一步提取特征。获得第二个卷积的特征后，特征的维度已经从$7\times32$，降低到了$5\times32$，因此可以直接使用求和（向量按位相加）的方式沿着电影名称维度进行降采样（$5\times32$ -> $1\times32$），得到最终的电影名称特征向量。
#
# 需要注意的是，降采样后的数据尺寸依然比下一层要求的输入向量多出一维 [2, 1, 32]，所以最终输出前需调整下形状。

# ##  4. 融合电影特征
#
与用户特征融合方式相同，电影特征融合采用特征级联加全连接层的方式，将电影特征用一个200维的向量表示。
```

```
# In[15]:


mov_combined = Linear(in_features=96, out_features=200)
# 收集所有的电影特征
_features = [mov_id_feat, mov_cat_feat, mov_title_feat]
_features = [k.numpy() for k in _features]
_features = [paddle.to_tensor(k) for k in _features]

# 对特征沿着最后一个维度级联
mov_feat = paddle.concat(_features, axis=1)
mov_feat = mov_combined(mov_feat)
mov_feat = F.tanh(mov_feat)
print("融合后的电影特征维度是：", mov_feat.shape)


#
至此已经完成了电影特征提取的网络设计，包括电影ID特征提取、电影类别特征提取和电影名称特征提取。
#
# 下面将这些模块整合到一个Python类中，完整代码如下：

# In[16]:


class MovModel(nn.Layer):
    def __init__(self, use_poster, use_mov_title, use_mov_cat, use_age_job,fc_sizes):
        super(MovModel, self).__init__()

        # 将传入的name信息和bool型参数添加到模型类中
        self.use_mov_poster = use_poster
        self.use_mov_title = use_mov_title
        self.use_usr_age_job = use_age_job
        self.use_mov_cat = use_mov_cat
        self.fc_sizes = fc_sizes

        # 获取数据集的信息，并构建训练和验证集的数据迭代器
        Dataset = MovieLen(self.use_mov_poster)
        self.Dataset = Dataset
        self.trainset = self.Dataset.train_dataset
        self.valset = self.Dataset.valid_dataset
        self.train_loader = self.Dataset.load_data(dataset=self.trainset, mode=
        'train')
        self.valid_loader = self.Dataset.load_data(dataset=self.valset, mode='valid')

        """ define network layer for embedding usr info """
        # 对电影ID信息做映射，并紧接着一个Linear层
        MOV_DICT_SIZE = Dataset.max_mov_id + 1
        self.mov_emb = Embedding(num_embeddings=MOV_DICT_SIZE, embedding_dim=32)
        self.mov_fc = Linear(32, 32)

        # 对电影类别做映射
        CATEGORY_DICT_SIZE = len(Dataset.movie_cat) + 1
        self.mov_cat_emb = Embedding(num_embeddings=CATEGORY_DICT_SIZE, embedding_dim
        =32)
        self.mov_cat_fc = Linear(32, 32)

        # 对电影名称做映射
        MOV_TITLE_DICT_SIZE = len(Dataset.movie_title) + 1
        self.mov_title_emb = Embedding(num_embeddings=MOV_TITLE_DICT_SIZE,
        embedding_dim=32)
        self.mov_title_conv = Conv2D(in_channels=1, out_channels=1, kernel_size=(3, 1
        ), stride=(2,1), padding=0)
        self.mov_title_conv2 = Conv2D(in_channels=1, out_channels=1, kernel_size=(3,
        1), stride=1, padding=0)

        # 新建一个Linear层，用于整合电影特征
        self.mov_concat_embed = Linear(in_features=96, out_features=200)

        #电影特征和用户特征使用了不同的全连接层，不共享参数
        movie_sizes = [200] + self.fc_sizes
```

```
            acts = ["relu" for _ in range(len(self.fc_sizes))]
            self._movie_layers = []
            for i in range(len(self.fc_sizes)):
                linear = Linear(
                    in_features=movie_sizes[i],
                    out_features=movie_sizes[i + 1],
                    weight_attr=paddle.ParamAttr(
                        initializer=nn.initializer.Normal(
                            std=1.0 / math.sqrt(movie_sizes[i]))))
                self.add_sublayer('linear_movie_%d' % i, linear)
                self._movie_layers.append(linear)
                if acts[i] == 'relu':
                    act = nn.ReLU()
                    self.add_sublayer('movie_act_%d' % i, act)
                    self._movie_layers.append(act)

        # 定义电影特征的前向计算过程
        def get_mov_feat(self, mov_var):
            """ get movie features"""
            # 获得电影数据
            mov_id, mov_cat, mov_title, mov_poster = mov_var
            feats_collect = []
            # 获得batchsize的大小
            batch_size = mov_id.shape[0]
            # 计算电影ID的特征，并存在feats_collect中
            mov_id = self.mov_emb(mov_id)
            mov_id = self.mov_fc(mov_id)
            mov_id = F.relu(mov_id)
            feats_collect.append(mov_id)

            # 如果使用电影的种类数据，计算电影种类特征的映射
            if self.use_mov_cat:
                # 计算电影种类的特征映射，对多个种类的特征求和得到最终特征
                mov_cat = self.mov_cat_emb(mov_cat)
                print(mov_title.shape)
                mov_cat = paddle.sum(mov_cat, axis=1, keepdim=False)

                mov_cat = self.mov_cat_fc(mov_cat)
                feats_collect.append(mov_cat)

            if self.use_mov_title:
                # 计算电影名字的特征映射，对特征映射使用卷积计算最终的特征
                mov_title = self.mov_title_emb(mov_title)
                mov_title = F.relu(self.mov_title_conv2(F.relu(self.mov_title_conv(
                mov_title))))

                mov_title = paddle.sum(mov_title, axis=2, keepdim=False)
                mov_title = F.relu(mov_title)
                mov_title = paddle.reshape(mov_title, [batch_size, -1])
                feats_collect.append(mov_title)

            # 使用一个全连接层，整合所有电影特征，映射为一个200维的特征向量
            mov_feat = paddle.concat(feats_collect, axis=1)
            mov_features = F.tanh(self.mov_concat_embed(mov_feat))
            for n_layer in self._movie_layers:
                mov_features = n_layer(mov_features)
            return mov_features


    # 由上述电影特征处理的代码可以观察到：
    # * 电影ID特征的计算方式和用户ID的计算方式相同。
    # *
    对于包含多个元素的电影类别数据，采用将所有元素的映射向量求和的结果,然后加上全连接结构作为最终的电影类别特征表示。考虑到电影类别的数量有限，这里采用简单的求和特征融合方式。
    # *
    对于电影的名称数据，其包含的元素数量多于电影种类元素数量，则采用卷积计算的方式，之后再将计算的特征沿着数据维度进行求和。读者也可自行设计这部分特征的计算网络，并观察最终训练结果。

    # 下面使用定义好的数据读取器，实现从电影数据中提取电影特征。

    # In[17]:
```

```
## 测试电影特征提取网络
fc_sizes=[128, 64, 32]
model = MovModel(use_poster=False, use_mov_title=True, use_mov_cat=True, use_age_job=True,fc_sizes=fc_sizes)
model.eval()

data_loader = model.train_loader

for idx, data in enumerate(data_loader()):
    # 获得数据，并转为动态图格式
    usr, mov, score = data
    # 只使用每个Batch的第一条数据
    mov_v = [var[0:1] for var in mov]

    _mov_v = [np.squeeze(var[0:1]) for var in mov]
    print("输入的电影ID数据：{}\n类别数据：{} \n名称数据：{} ".format(*_mov_v))
    mov_v = [paddle.to_tensor(var) for var in mov_v]
    mov_feat = model.get_mov_feat(mov_v)
    print("计算得到的电影特征维度是：", mov_feat.shape)
    break



# # 相似度计算
#
#
计算得到用户特征和电影特征后，我们还需要计算特征之间的相似度。如果一个用户对某个电影很感兴趣，并给了五分评价，那么该用户和电影特征之间的相似度是很高的。
#
#
衡量向量距离（相似度）有多种方案：欧式距离、曼哈顿距离、切比雪夫距离、余弦相似度等，本节我们使用忽略尺度信息的余弦相似度构建相似度矩阵。余弦相似度又称为余弦相似性，是通过计算两个向量的夹角余弦值来评估他们的相似度，如下图，两条红色的直线表示两个向量，之间的夹角可以用来表示相似度大小，角度为0时，余弦值为1，表示完全相似。
#
# <img src="
https://ai-studio-static-online.cdn.bcebos.com/7d955048899441aeade18be12ae5a21c2be3b0f6a3e04374a595ba73801eef82"
# width="300" >
#
#
# 余弦相似度的公式为：
#
# $similarity = cos(\theta) = \frac{A\cdot B}{A + B} = \frac{\sum_{i}^{n}A_i \times B_i}{\sqrt{\sum_{i}^{n}(A_i)^2 + \sum_{i}^{n}(B_i)^2}}$
#
#
#
下面是计算相似度的实现方法，输入用户特征和电影特征，计算出两者之间的相似度。另外，我们将用户对电影的评分作为相似度衡量的标准，由于相似度的数据范围是[0,
1]，还需要把计算的相似度扩大到评分数据范围，评分分为1-5共5个档次，所以需要将相似度扩大5倍。使用飞桨[scale API](
http://www.paddlepaddle.org.cn/documentation/docs/zh/2.0-rc/api/paddle/fluid/layers/scale_cn.html#scale)，可以对输入数据进行缩放。计算余弦相似度可以使用[cosine_similarity API](
https://https://www.paddlepaddle.org.cn/documentation/docs/zh/2.0-rc/api/paddle/nn/functional/common/cosine_similarity_cn.html#cosine-similarity) 完成。

# In[18]:


def similarty(usr_feature, mov_feature):
    res = F.cosine_similarity(usr_feature, mov_feature)
    res = paddle.scale(res, scale=5)
    return usr_feat, mov_feat, res

# 使用上文计算得到的用户特征和电影特征计算相似度
usr_feat, mov_feat, _sim = similarty(usr_feat, mov_feat)
print("相似度得分是：", np.squeeze(_sim.numpy()))
```

```
# 从结果中我们发现相似度很小，主要有以下原因：
# 1. 神经网络并没有训练，模型参数都是随机初始化的，提取出的特征没有规律性。
# 2. 计算相似度的用户数据和电影数据相关性很小。
#
# 下一节我们就开始训练，让这个网络能够输出有效的用户特征向量和电影特征向量。

# ## 总结
#
#
本节中，我们介绍了个性化推荐的模型设计，包括用户特征网络、电影特征网络和特征相似度计算三部分。
#
#
其中，用户特征网络将用户数据映射为固定长度的特征向量，电影特征网络将电影数据映射为固定长度的特征向量，最终利用余弦相似度计算出用户特征和电影特征的相似度。相似度越大，表示用户对该电影越喜欢。
#
# 以下为模型设计的完整代码：

# In[19]:


class Model(nn.Layer):
    def __init__(self, use_poster, use_mov_title, use_mov_cat, use_age_job):
        super(Model, self).__init__()

        # 将传入的name信息和bool型参数添加到模型类中
        self.use_mov_poster = use_poster
        self.use_mov_title = use_mov_title
        self.use_usr_age_job = use_age_job
        self.use_mov_cat = use_mov_cat

        # 获取数据集的信息，并构建训练和验证集的数据迭代器
        Dataset = MovieLen(self.use_mov_poster)
        self.Dataset = Dataset
        self.trainset = self.Dataset.train_dataset
        self.valset = self.Dataset.valid_dataset
        self.train_loader = self.Dataset.load_data(dataset=self.trainset, mode=
        'train')
        self.valid_loader = self.Dataset.load_data(dataset=self.valset, mode='valid')

        """ define network layer for embedding usr info """
        USR_ID_NUM = Dataset.max_usr_id + 1
        # 对用户ID做映射，并紧接着一个Linear层
        self.usr_emb = Embedding(num_embeddings=USR_ID_NUM, embedding_dim=32, sparse=
        False)
        self.usr_fc = Linear(in_features=32, out_features=32)

        # 对用户性别信息做映射，并紧接着一个Linear层
        USR_GENDER_DICT_SIZE = 2
        self.usr_gender_emb = Embedding(num_embeddings=USR_GENDER_DICT_SIZE,
        embedding_dim=16)
        self.usr_gender_fc = Linear(in_features=16, out_features=16)

        # 对用户年龄信息做映射，并紧接着一个Linear层
        USR_AGE_DICT_SIZE = Dataset.max_usr_age + 1
        self.usr_age_emb = Embedding(num_embeddings=USR_AGE_DICT_SIZE, embedding_dim=
        16)
        self.usr_age_fc = Linear(in_features=16, out_features=16)

        # 对用户职业信息做映射，并紧接着一个Linear层
        USR_JOB_DICT_SIZE = Dataset.max_usr_job + 1
        self.usr_job_emb = Embedding(num_embeddings=USR_JOB_DICT_SIZE, embedding_dim=
        16)
        self.usr_job_fc = Linear(in_features=16, out_features=16)

        # 新建一个Linear层，用于整合用户数据信息
        self.usr_combined = Linear(in_features=80, out_features=200)

        """ define network layer for embedding usr info """
        # 对电影ID信息做映射，并紧接着一个Linear层
```

```
        MOV_DICT_SIZE = Dataset.max_mov_id + 1
        self.mov_emb = Embedding(num_embeddings=MOV_DICT_SIZE, embedding_dim=32)
        self.mov_fc = Linear(in_features=32, out_features=32)

        # 对电影类别做映射
        CATEGORY_DICT_SIZE = len(Dataset.movie_cat) + 1
        self.mov_cat_emb = Embedding(num_embeddings=CATEGORY_DICT_SIZE, embedding_dim
        =32, sparse=False)
        self.mov_cat_fc = Linear(in_features=32, out_features=32)

        # 对电影名称做映射
        MOV_TITLE_DICT_SIZE = len(Dataset.movie_title) + 1
        self.mov_title_emb = Embedding(num_embeddings=MOV_TITLE_DICT_SIZE,
        embedding_dim=32, sparse=False)
        self.mov_title_conv = Conv2D(in_channels=1, out_channels=1, kernel_size=(3, 1
        ), stride=(2,1), padding=0)
        self.mov_title_conv2 = Conv2D(in_channels=1, out_channels=1, kernel_size=(3,
        1), stride=1, padding=0)

        # 新建一个FC层，用于整合电影特征
        self.mov_concat_embed = Linear(in_features=96, out_features=200)

        user_sizes = [200] + self.fc_sizes
        acts = ["relu" for _ in range(len(self.fc_sizes))]
        self._user_layers = []
        for i in range(len(self.fc_sizes)):
            linear = Linear(
                in_features=user_sizes[i],
                out_features=user_sizes[i + 1],
                weight_attr=paddle.ParamAttr(
                    initializer=nn.initializer.Normal(
                        std=1.0 / math.sqrt(user_sizes[i])))))
            self.add_sublayer('linear_user_%d' % i, linear)
            self._user_layers.append(linear)
            if acts[i] == 'relu':
                act = nn.ReLU()
                self.add_sublayer('user_act_%d' % i, act)
                self._user_layers.append(act)

        #电影特征和用户特征使用了不同的全连接层，不共享参数
        movie_sizes = [200] + self.fc_sizes
        acts = ["relu" for _ in range(len(self.fc_sizes))]
        self._movie_layers = []
        for i in range(len(self.fc_sizes)):
            linear = nn.Linear(
                in_features=movie_sizes[i],
                out_features=movie_sizes[i + 1],
                weight_attr=paddle.ParamAttr(
                    initializer=nn.initializer.Normal(
                        std=1.0 / math.sqrt(movie_sizes[i])))))
            self.add_sublayer('linear_movie_%d' % i, linear)
            self._movie_layers.append(linear)
            if acts[i] == 'relu':
                act = nn.ReLU()
                self.add_sublayer('movie_act_%d' % i, act)
                self._movie_layers.append(act)

    # 定义计算用户特征的前向运算过程
    def get_usr_feat(self, usr_var):
        """ get usr features"""
        # 获取到用户数据
        usr_id, usr_gender, usr_age, usr_job = usr_var
        # 将用户的ID数据经过embedding和Linear计算，得到的特征保存在feats_collect中
        feats_collect = []
        usr_id = self.usr_emb(usr_id)
        usr_id = self.usr_fc(usr_id)
        usr_id = F.relu(usr_id)
        feats_collect.append(usr_id)

        # 计算用户的性别特征，并保存在feats_collect中
        usr_gender = self.usr_gender_emb(usr_gender)
        usr_gender = self.usr_gender_fc(usr_gender)
```

```
            usr_gender = F.relu(usr_gender)
            feats_collect.append(usr_gender)
            # 选择是否使用用户的年龄-职业特征
            if self.use_usr_age_job:
                # 计算用户的年龄特征，并保存在feats_collect中
                usr_age = self.usr_age_emb(usr_age)
                usr_age = self.usr_age_fc(usr_age)
                usr_age = F.relu(usr_age)
                feats_collect.append(usr_age)
                # 计算用户的职业特征，并保存在feats_collect中
                usr_job = self.usr_job_emb(usr_job)
                usr_job = self.usr_job_fc(usr_job)
                usr_job = F.relu(usr_job)
                feats_collect.append(usr_job)

            # 将用户的特征级联，并通过Linear层得到最终的用户特征
            usr_feat = paddle.concat(feats_collect, axis=1)
            user_features = F.tanh(self.usr_combined(usr_feat))
            #通过3层全链接层，获得用于计算相似度的用户特征和电影特征
            for n_layer in self._user_layers:
                user_features = n_layer(user_features)

            return user_features

        # 定义电影特征的前向计算过程
        def get_mov_feat(self, mov_var):
            """ get movie features"""
            # 获得电影数据
            mov_id, mov_cat, mov_title, mov_poster = mov_var
            feats_collect = []
            # 获得batchsize的大小
            batch_size = mov_id.shape[0]
            # 计算电影ID的特征，并存在feats_collect中
            mov_id = self.mov_emb(mov_id)
            mov_id = self.mov_fc(mov_id)
            mov_id = F.relu(mov_id)
            feats_collect.append(mov_id)

            # 如果使用电影的种类数据，计算电影种类特征的映射
            if self.use_mov_cat:
                # 计算电影种类的特征映射，对多个种类的特征求和得到最终特征
                mov_cat = self.mov_cat_emb(mov_cat)
                mov_cat = paddle.sum(mov_cat, axis=1, keepdim=False)

                mov_cat = self.mov_cat_fc(mov_cat)
                feats_collect.append(mov_cat)

            if self.use_mov_title:
                # 计算电影名字的特征映射，对特征映射使用卷积计算最终的特征
                mov_title = self.mov_title_emb(mov_title)
                mov_title = F.relu(self.mov_title_conv2(F.relu(self.mov_title_conv(
                mov_title))))
                mov_title = paddle.sum(mov_title, axis=2, keepdim=False)
                mov_title = F.relu(mov_title)
                mov_title = paddle.reshape(mov_title, [batch_size, -1])
                feats_collect.append(mov_title)

            # 使用一个全连接层，整合所有电影特征，映射为一个200维的特征向量
            mov_feat = paddle.concat(feats_collect, axis=1)
            mov_features = F.tanh(self.mov_concat_embed(mov_feat))

            for n_layer in self._movie_layers:
                mov_features = n_layer(mov_features)

            return mov_features

        # 定义个性化推荐算法的前向计算
        def forward(self, usr_var, mov_var):
            # 计算用户特征和电影特征
            usr_feat = self.get_usr_feat(usr_var)
            mov_feat = self.get_mov_feat(mov_var)

```

```
        #通过3层全连接层，获得用于计算相似度的用户特征和电影特征
        for n_layer in self._user_layers:
            user_features = n_layer(user_features)

        for n_layer in self._movie_layers:
            mov_features = n_layer(mov_features)

        # 根据计算的特征计算相似度
        res = F.cosine_similarity(user_features, mov_features)
        # 将相似度扩大范围到和电影评分相同数据范围
        res = paddle.scale(res, scale=5)
        return usr_feat, mov_feat, res


# train.py
#!/usr/bin/env python
# coding: utf-8

# 启动训练前，复用前面章节的数据处理和神经网络模型代码，已阅读可直接跳过。
#

# In[ ]:


import random
import numpy as np
from PIL import Image

import paddle
from paddle.nn import Linear, Embedding, Conv2D
import paddle.nn.functional as F
import math

class MovieLen(object):
    def __init__(self, use_poster):
        self.use_poster = use_poster
        # 声明每个数据文件的路径
        usr_info_path = "./work/ml-1m/users.dat"
        if use_poster:
            rating_path = "./work/ml-1m/new_rating.txt"
        else:
            rating_path = "./work/ml-1m/ratings.dat"

        movie_info_path = "./work/ml-1m/movies.dat"
        self.poster_path = "./work/ml-1m/posters/"
        # 得到电影数据
        self.movie_info, self.movie_cat, self.movie_title = self.get_movie_info(
        movie_info_path)
        # 记录电影的最大ID
        self.max_mov_cat = np.max([self.movie_cat[k] for k in self.movie_cat])
        self.max_mov_tit = np.max([self.movie_title[k] for k in self.movie_title])
        self.max_mov_id = np.max(list(map(int, self.movie_info.keys())))
        # 记录用户数据的最大ID
        self.max_usr_id = 0
        self.max_usr_age = 0
        self.max_usr_job = 0
        # 得到用户数据
        self.usr_info = self.get_usr_info(usr_info_path)
        # 得到评分数据
        self.rating_info = self.get_rating_info(rating_path)
        # 构建数据集
        self.dataset = self.get_dataset(usr_info=self.usr_info,
                                        rating_info=self.rating_info,
                                        movie_info=self.movie_info)
        # 划分数据集，获得数据加载器
        self.train_dataset = self.dataset[:int(len(self.dataset)*0.9)]
        self.valid_dataset = self.dataset[int(len(self.dataset)*0.9):]
        print("##Total dataset instances: ", len(self.dataset))
        print("##MovieLens dataset information: \nusr num: {}\n"
              "movies num: {}".format(len(self.usr_info),len(self.movie_info)))
    # 得到电影数据
    def get_movie_info(self, path):
```

```
            # 打开文件，编码方式选择ISO-8859-1，读取所有数据到data中
            with open(path, 'r', encoding="ISO-8859-1") as f:
                data = f.readlines()
            # 建立三个字典，分别用户存放电影所有信息，电影的名字信息、类别信息
            movie_info, movie_titles, movie_cat = {}, {}, {}
            # 对电影名字、类别中不同的单词计数
            t_count, c_count = 1, 1

            count_tit = {}
            # 按行读取数据并处理
            for item in data:
                item = item.strip().split("::")
                v_id = item[0]
                v_title = item[1][:-7]
                cats = item[2].split('|')
                v_year = item[1][-5:-1]

                titles = v_title.split()
                # 统计电影名字的单词，并给每个单词一个序号，放在movie_titles中
                for t in titles:
                    if t not in movie_titles:
                        movie_titles[t] = t_count
                        t_count += 1
                # 统计电影类别单词，并给每个单词一个序号，放在movie_cat中
                for cat in cats:
                    if cat not in movie_cat:
                        movie_cat[cat] = c_count
                        c_count += 1
                # 补0使电影名称对应的列表长度为15
                v_tit = [movie_titles[k] for k in titles]
                while len(v_tit)<15:
                    v_tit.append(0)
                # 补0使电影种类对应的列表长度为6
                v_cat = [movie_cat[k] for k in cats]
                while len(v_cat)<6:
                    v_cat.append(0)
                # 保存电影数据到movie_info中
                movie_info[v_id] = {'mov_id': int(v_id),
                                    'title': v_tit,
                                    'category': v_cat,
                                    'years': int(v_year)}
            return movie_info, movie_cat, movie_titles

        def get_usr_info(self, path):
            # 性别转换函数，M-0，  F-1
            def gender2num(gender):
                return 1 if gender == 'F' else 0

            # 打开文件，读取所有行到data中
            with open(path, 'r') as f:
                data = f.readlines()
            # 建立用户信息的字典
            use_info = {}

            max_usr_id = 0
            #按行索引数据
            for item in data:
                # 去除每一行中和数据无关的部分
                item = item.strip().split("::")
                usr_id = item[0]
                # 将字符数据转成数字并保存在字典中
                use_info[usr_id] = {'usr_id': int(usr_id),
                                    'gender': gender2num(item[1]),
                                    'age': int(item[2]),
                                    'job': int(item[3])}
                self.max_usr_id = max(self.max_usr_id, int(usr_id))
                self.max_usr_age = max(self.max_usr_age, int(item[2]))
                self.max_usr_job = max(self.max_usr_job, int(item[3]))
            return use_info
        # 得到评分数据
        def get_rating_info(self, path):
            # 读取文件里的数据
```

```
            with open(path, 'r') as f:
                data = f.readlines()
            # 将数据保存在字典中并返回
            rating_info = {}
            for item in data:
                item = item.strip().split("::")
                usr_id,movie_id,score = item[0],item[1],item[2]
                if usr_id not in rating_info.keys():
                    rating_info[usr_id] = {movie_id:float(score)}
                else:
                    rating_info[usr_id][movie_id] = float(score)
            return rating_info
        # 构建数据集
        def get_dataset(self, usr_info, rating_info, movie_info):
            trainset = []
            for usr_id in rating_info.keys():
                usr_ratings = rating_info[usr_id]
                for movie_id in usr_ratings:
                    trainset.append({'usr_info': usr_info[usr_id],
                                     'mov_info': movie_info[movie_id],
                                     'scores': usr_ratings[movie_id]})
            return trainset

        def load_data(self, dataset=None, mode='train'):
            use_poster = False

            # 定义数据迭代Batch大小
            BATCHSIZE = 256

            data_length = len(dataset)
            index_list = list(range(data_length))
            # 定义数据迭代加载器
            def data_generator():
                # 训练模式下，打乱训练数据
                if mode == 'train':
                    random.shuffle(index_list)
                # 声明每个特征的列表
                usr_id_list,usr_gender_list,usr_age_list,usr_job_list = [], [], [], []
                mov_id_list,mov_tit_list,mov_cat_list,mov_poster_list = [], [], [], []
                score_list = []
                # 索引遍历输入数据集
                for idx, i in enumerate(index_list):
                    # 获得特征数据保存到对应特征列表中
                    usr_id_list.append(dataset[i]['usr_info']['usr_id'])
                    usr_gender_list.append(dataset[i]['usr_info']['gender'])
                    usr_age_list.append(dataset[i]['usr_info']['age'])
                    usr_job_list.append(dataset[i]['usr_info']['job'])

                    mov_id_list.append(dataset[i]['mov_info']['mov_id'])
                    mov_tit_list.append(dataset[i]['mov_info']['title'])
                    mov_cat_list.append(dataset[i]['mov_info']['category'])
                    mov_id = dataset[i]['mov_info']['mov_id']

                    if use_poster:
                        # 不使用图像特征时，不读取图像数据，加快数据读取速度
                        poster = Image.open(self.poster_path+'mov_id{}.jpg'.format(str(
                        mov_id[0])))
                        poster = poster.resize([64, 64])
                        if len(poster.size) <= 2:
                            poster = poster.convert("RGB")

                        mov_poster_list.append(np.array(poster))

                    score_list.append(int(dataset[i]['scores']))
                    # 如果读取的数据量达到当前的batch大小，就返回当前批次
                    if len(usr_id_list)==BATCHSIZE:
                        # 转换列表数据为数组形式，reshape到固定形状
                        usr_id_arr = np.array(usr_id_list)
                        usr_gender_arr = np.array(usr_gender_list)
                        usr_age_arr = np.array(usr_age_list)
                        usr_job_arr = np.array(usr_job_list)
```

```
                    mov_id_arr = np.array(mov_id_list)
                    mov_cat_arr = np.reshape(np.array(mov_cat_list), [BATCHSIZE, 6]).
                    astype(np.int64)
                    mov_tit_arr = np.reshape(np.array(mov_tit_list), [BATCHSIZE, 1,
                    15]).astype(np.int64)

                    if use_poster:
                        mov_poster_arr = np.reshape(np.array(mov_poster_list)/127.5 -
                         1, [BATCHSIZE, 3, 64, 64]).astype(np.float32)
                    else:
                        mov_poster_arr = np.array([0.])

                    scores_arr = np.reshape(np.array(score_list), [-1, 1]).astype(np.
                    float32)

                    # 放回当前批次数据
                    yield [usr_id_arr, usr_gender_arr, usr_age_arr, usr_job_arr],
                                                [mov_id_arr, mov_cat_arr, mov_tit_arr
                    , mov_poster_arr], scores_arr

                    # 清空数据
                    usr_id_list, usr_gender_list, usr_age_list, usr_job_list = [],
                    [], [], []
                    mov_id_list, mov_tit_list, mov_cat_list, score_list = [], [], [],
                     []
                    mov_poster_list = []
        return data_generator


# In[ ]:


class Model(paddle.nn.Layer):
    def __init__(self, use_poster, use_mov_title, use_mov_cat, use_age_job,fc_sizes):
        super(Model, self).__init__()

        # 将传入的name信息和bool型参数添加到模型类中
        self.use_mov_poster = use_poster
        self.use_mov_title = use_mov_title
        self.use_usr_age_job = use_age_job
        self.use_mov_cat = use_mov_cat
        self.fc_sizes=fc_sizes

        # 获取数据集的信息，并构建训练和验证集的数据迭代器
        Dataset = MovieLen(self.use_mov_poster)
        self.Dataset = Dataset
        self.trainset = self.Dataset.train_dataset
        self.valset = self.Dataset.valid_dataset
        self.train_loader = self.Dataset.load_data(dataset=self.trainset, mode=
        'train')
        self.valid_loader = self.Dataset.load_data(dataset=self.valset, mode='valid')

        usr_embedding_dim=32
        gender_embeding_dim=16
        age_embedding_dim=16

        job_embedding_dim=16
        mov_embedding_dim=16
        category_embedding_dim=16
        title_embedding_dim=32

        """ define network layer for embedding usr info """
        USR_ID_NUM = Dataset.max_usr_id + 1

        # 对用户ID做映射，并紧接着一个Linear层
        self.usr_emb = Embedding(num_embeddings=USR_ID_NUM, embedding_dim=
        usr_embedding_dim, sparse=False)
        self.usr_fc = Linear(in_features=usr_embedding_dim, out_features=32)

        # 对用户性别信息做映射，并紧接着一个Linear层
        USR_GENDER_DICT_SIZE = 2
        self.usr_gender_emb = Embedding(num_embeddings=USR_GENDER_DICT_SIZE,
```

```
        embedding_dim=gender_embeding_dim)
1554    self.usr_gender_fc = Linear(in_features=gender_embeding_dim, out_features=16)
1555
1556    # 对用户年龄信息做映射，并紧接着一个Linear层
1557    USR_AGE_DICT_SIZE = Dataset.max_usr_age + 1
1558    self.usr_age_emb = Embedding(num_embeddings=USR_AGE_DICT_SIZE, embedding_dim=
        age_embedding_dim)
1559    self.usr_age_fc = Linear(in_features=age_embedding_dim, out_features=16)
1560
1561    # 对用户职业信息做映射，并紧接着一个Linear层
1562    USR_JOB_DICT_SIZE = Dataset.max_usr_job + 1
1563    self.usr_job_emb = Embedding(num_embeddings=USR_JOB_DICT_SIZE, embedding_dim=
        job_embedding_dim)
1564    self.usr_job_fc = Linear(in_features=job_embedding_dim, out_features=16)
1565
1566    # 新建一个Linear层，用于整合用户数据信息
1567    self.usr_combined = Linear(in_features=80, out_features=200)
1568
1569    """ define network layer for embedding usr info """
1570    # 对电影ID信息做映射，并紧接着一个Linear层
1571    MOV_DICT_SIZE = Dataset.max_mov_id + 1
1572    self.mov_emb = Embedding(num_embeddings=MOV_DICT_SIZE, embedding_dim=
        mov_embedding_dim)
1573    self.mov_fc = Linear(in_features=mov_embedding_dim, out_features=32)
1574
1575    # 对电影类别做映射
1576    CATEGORY_DICT_SIZE = len(Dataset.movie_cat) + 1
1577    self.mov_cat_emb = Embedding(num_embeddings=CATEGORY_DICT_SIZE, embedding_dim
        =category_embedding_dim, sparse=False)
1578    self.mov_cat_fc = Linear(in_features=category_embedding_dim, out_features=32)
1579
1580    # 对电影名称做映射
1581    MOV_TITLE_DICT_SIZE = len(Dataset.movie_title) + 1
1582    self.mov_title_emb = Embedding(num_embeddings=MOV_TITLE_DICT_SIZE,
        embedding_dim=title_embedding_dim, sparse=False)
1583    self.mov_title_conv = Conv2D(in_channels=1, out_channels=1, kernel_size=(3, 1
        ), stride=(2,1), padding=0)
1584    self.mov_title_conv2 = Conv2D(in_channels=1, out_channels=1, kernel_size=(3,
        1), stride=1, padding=0)
1585
1586    # 新建一个Linear层，用于整合电影特征
1587    self.mov_concat_embed = Linear(in_features=96, out_features=200)
1588
1589    user_sizes = [200] + self.fc_sizes
1590    acts = ["relu" for _ in range(len(self.fc_sizes))]
1591    self._user_layers = []
1592    for i in range(len(self.fc_sizes)):
1593        linear = paddle.nn.Linear(
1594            in_features=user_sizes[i],
1595            out_features=user_sizes[i + 1],
1596            weight_attr=paddle.ParamAttr(
1597                initializer=paddle.nn.initializer.Normal(
1598                    std=1.0 / math.sqrt(user_sizes[i]))))
1599        self.add_sublayer('linear_user_%d' % i, linear)
1600        self._user_layers.append(linear)
1601        if acts[i] == 'relu':
1602            act = paddle.nn.ReLU()
1603            self.add_sublayer('user_act_%d' % i, act)
1604            self._user_layers.append(act)
1605
1606    #电影特征和用户特征使用了不同的全连接层，不共享参数
1607    movie_sizes = [200] + self.fc_sizes
1608    acts = ["relu" for _ in range(len(self.fc_sizes))]
1609    self._movie_layers = []
1610    for i in range(len(self.fc_sizes)):
1611        linear = paddle.nn.Linear(
1612            in_features=movie_sizes[i],
1613            out_features=movie_sizes[i + 1],
1614            weight_attr=paddle.ParamAttr(
1615                initializer=paddle.nn.initializer.Normal(
1616                    std=1.0 / math.sqrt(movie_sizes[i]))))
1617        self.add_sublayer('linear_movie_%d' % i, linear)
```

```
            self._movie_layers.append(linear)
            if acts[i] == 'relu':
                act = paddle.nn.ReLU()
                self.add_sublayer('movie_act_%d' % i, act)
                self._movie_layers.append(act)

    # 定义计算用户特征的前向运算过程
    def get_usr_feat(self, usr_var):
        """ get usr features"""
        # 获取到用户数据
        usr_id, usr_gender, usr_age, usr_job = usr_var
        # 将用户的ID数据经过embedding和Linear计算，得到的特征保存在feats_collect中
        feats_collect = []
        usr_id = self.usr_emb(usr_id)
        usr_id = self.usr_fc(usr_id)
        usr_id = F.relu(usr_id)
        feats_collect.append(usr_id)

        # 计算用户的性别特征，并保存在feats_collect中
        usr_gender = self.usr_gender_emb(usr_gender)
        usr_gender = self.usr_gender_fc(usr_gender)
        usr_gender = F.relu(usr_gender)
        feats_collect.append(usr_gender)
        # 选择是否使用用户的年龄-职业特征
        if self.use_usr_age_job:
            # 计算用户的年龄特征，并保存在feats_collect中
            usr_age = self.usr_age_emb(usr_age)
            usr_age = self.usr_age_fc(usr_age)
            usr_age = F.relu(usr_age)
            feats_collect.append(usr_age)
            # 计算用户的职业特征，并保存在feats_collect中
            usr_job = self.usr_job_emb(usr_job)
            usr_job = self.usr_job_fc(usr_job)
            usr_job = F.relu(usr_job)
            feats_collect.append(usr_job)

        # 将用户的特征级联，并通过Linear层得到最终的用户特征
        usr_feat = paddle.concat(feats_collect, axis=1)
        user_features = F.tanh(self.usr_combined(usr_feat))

        #通过3层全链接层，获得用于计算相似度的用户特征和电影特征
        for n_layer in self._user_layers:
            user_features = n_layer(user_features)

        return user_features

        # 定义电影特征的前向计算过程
    def get_mov_feat(self, mov_var):
        """ get movie features"""
        # 获得电影数据
        mov_id, mov_cat, mov_title, mov_poster = mov_var
        feats_collect = []
        # 获得batchsize的大小
        batch_size = mov_id.shape[0]
        # 计算电影ID的特征，并存在feats_collect中
        mov_id = self.mov_emb(mov_id)
        mov_id = self.mov_fc(mov_id)
        mov_id = F.relu(mov_id)
        feats_collect.append(mov_id)

        # 如果使用电影的种类数据，计算电影种类特征的映射
        if self.use_mov_cat:
            # 计算电影种类的特征映射，对多个种类的特征求和得到最终特征
            mov_cat = self.mov_cat_emb(mov_cat)
            mov_cat = paddle.sum(mov_cat, axis=1, keepdim=False)

            mov_cat = self.mov_cat_fc(mov_cat)
            feats_collect.append(mov_cat)

        if self.use_mov_title:
            # 计算电影名字的特征映射，对特征映射使用卷积计算最终的特征
            mov_title = self.mov_title_emb(mov_title)
```

```
            mov_title = F.relu(self.mov_title_conv2(F.relu(self.mov_title_conv(
            mov_title))))
            mov_title = paddle.sum(mov_title, axis=2, keepdim=False)
            mov_title = F.relu(mov_title)
            mov_title = paddle.reshape(mov_title, [batch_size, -1])

            feats_collect.append(mov_title)

        # 使用一个全连接层，整合所有电影特征，映射为一个200维的特征向量
        mov_feat = paddle.concat(feats_collect, axis=1)
        mov_features = F.tanh(self.mov_concat_embed(mov_feat))

        for n_layer in self._movie_layers:
            mov_features = n_layer(mov_features)

        return mov_features

    # 定义个性化推荐算法的前向计算
    def forward(self, usr_var, mov_var):
        # 计算用户特征和电影特征
        user_features = self.get_usr_feat(usr_var)
        mov_features = self.get_mov_feat(mov_var)

        # 根据计算的特征计算相似度
        sim = F.common.cosine_similarity(user_features, mov_features).reshape([-1, 1])
        #使用余弦相似度算子，计算用户和电影的相似程度
        # sim = F.cosine_similarity(user_features, mov_features,
        axis=1).reshape([-1, 1])
        # 将相似度扩大范围到和电影评分相同数据范围
        res = paddle.scale(sim, scale=5)
        return user_features, mov_features, res



# In[ ]:


# 解压数据集
get_ipython().system('unzip -o -q -d ~/work/ ~/data/data19736/ml-1m.zip')


# # 模型训练
#
#
在模型训练前需要定义好训练的参数，包括是否使用GPU、设置损失函数、选择优化器以及学习率等。
#
在本次任务中，由于数据较为简单，我们选择在CPU上训练，优化器使用Adam，学习率设置为0.01，一共训练5个epoch。
#
#
然而，针对推荐算法的网络，如何设置损失函数呢？在CV和NLP章节中的案例多是分类问题，采用交叉熵作为损失函数。但在电影推荐中，可以作为标签的只有评分数据，因此，我们用评分数据作为监督信息，神经网络的输出作为预测值，使用均方差（Mean Square
Error）损失函数去训练网络模型。
#
# ><font
size=2>说明：使用均方差损失函数即使用回归的方法完成模型训练。电影的评分数据只有5个，是否可以使用分类损失函数完成训练呢？事实上，评分数据是一个连续数据，如评分3和评分4是接近的，如果使用分类的方法，评分3和评分4是两个类别，容易割裂评分间的连续性。
#
#
很多互联网产品会以用户的点击或消费数据作为训练数据，这些数据是二分类问题（点或不点，买或不买），可以采用交叉熵等分类任务的损失函数。
# </font>
#
# 整个训练过程和其他的模型训练大同小异，不再赘述。

# In[ ]:


def train(model):
```

```
    # 配置训练参数
    lr = 0.001
    Epoches = 10
    paddle.set_device('cpu')

    # 启动训练
    model.train()
    # 获得数据读取器
    data_loader = model.train_loader
    # 使用adam优化器，学习率使用0.01
    opt = paddle.optimizer.Adam(learning_rate=lr, parameters=model.parameters())

    for epoch in range(0, Epoches):
        for idx, data in enumerate(data_loader()):
            # 获得数据，并转为tensor格式
            usr, mov, score = data
            usr_v = [paddle.to_tensor(var) for var in usr]
            mov_v = [paddle.to_tensor(var) for var in mov]
            scores_label = paddle.to_tensor(score)
            # 计算出算法的前向计算结果
            _, _, scores_predict = model(usr_v, mov_v)
            # 计算loss
            loss = F.square_error_cost(scores_predict, scores_label)
            avg_loss = paddle.mean(loss)

            if idx % 500 == 0:
                print("epoch: {}, batch_id: {}, loss is: {}".format(epoch, idx,
                avg_loss.numpy()))

            # 损失函数下降，并清除梯度
            avg_loss.backward()
            opt.step()
            opt.clear_grad()

        # 每个epoch 保存一次模型
        paddle.save(model.state_dict(), './checkpoint/epoch'+str(epoch)+'.pdparams')


# In[ ]:


# 启动训练
fc_sizes=[128, 64, 32]
use_poster, use_mov_title, use_mov_cat, use_age_job = False, True, True, True
model = Model(use_poster, use_mov_title, use_mov_cat, use_age_job,fc_sizes)
train(model)


#
从训练结果来看，Loss保持在1以下的范围，主要是因为使用的均方差Loss，计算得到预测评分和真实评分的均方差，真实评分的数据是1-5之间的整数，评分数据较大导致计算出来的Loss也偏大。
#
# 不过不用担心，我们只是通过训练神经网络提取特征向量，Loss只要收敛即可。

# 对训练的模型在验证集上做评估，除了训练所使用的Loss之外，还有两个选择：
# 1.
评分预测精度ACC(Accuracy)：将预测的float数字转成整数，计算预测评分和真实评分的匹配度。评分误差在0.5分以内的算正确，否则算错误。
# 2. 评分预测误差（Mean Absolut Error）MAE：计算预测评分和真实评分之间的平均绝对误差。
# 3. 均方根误差 （Root Mean Squard Error）RMSE：计算预测评分和真实值之间的平均平方误差
#
# 下面是使用训练集评估这两个指标的代码实现。

# In[ ]:


from math import sqrt
def evaluation(model, params_file_path):
    model_state_dict = paddle.load(params_file_path)
    model.load_dict(model_state_dict)
    model.eval()
```

```
    acc_set = []
    avg_loss_set = []
    squaredError=[]
    for idx, data in enumerate(model.valid_loader()):
        usr, mov, score_label = data
        usr_v = [paddle.to_tensor(var) for var in usr]
        mov_v = [paddle.to_tensor(var) for var in mov]

        _, _, scores_predict = model(usr_v, mov_v)

        pred_scores = scores_predict.numpy()

        avg_loss_set.append(np.mean(np.abs(pred_scores - score_label)))
        squaredError.extend(np.abs(pred_scores - score_label)**2)

        diff = np.abs(pred_scores - score_label)
        diff[diff>0.5] = 1
        acc = 1 - np.mean(diff)
        acc_set.append(acc)
    RMSE=sqrt(np.sum(squaredError) / len(squaredError))
    # print("RMSE = ", sqrt(np.sum(squaredError) / len(squaredError)))#均方根误差RMSE
    return np.mean(acc_set), np.mean(avg_loss_set),RMSE


# In[ ]:


param_path = "./checkpoint/epoch"
for i in range(10):
    acc, mae,RMSE = evaluation(model, param_path+str(i)+'.pdparams')
    print("ACC:", acc, "MAE:", mae,'RMSE:',RMSE)


#
上述结果中，我们采用了ACC和MAE指标测试在验证集上的评分预测的准确性，其中ACC值越大越好，MAE值越小越好，RMSE越小也越好。
#
# ><font
size=2>可以看到ACC和MAE的值不是很理想，但是这仅仅是对于评分预测不准确，不能直接衡量推荐结果的准确性。考虑到我们设计的神经网络是为了完成推荐任务而不是评分任务，所以：
# <br>1.
只针对预测评分任务来说，我们设计的模型不够合理或者训练数据不足，导致评分预测不理想；
# <br>2.
从损失函数的收敛可以知道网络的训练是有效的，但评分预测的好坏不能完全反映推荐结果的好坏。</font>
#
#
到这里，我们已经完成了推荐算法的前三步，包括：数据的准备、神经网络的设计和神经网络的训练。
#
# 目前还需要完成剩余的两个步骤：
#
# 1. 提取用户、电影数据的特征并保存到本地；
#
# 2. 利用保存的特征计算相似度矩阵，利用相似度完成推荐。
#
#
下面，我们利用训练的神经网络提取数据的特征，进而完成电影推荐，并观察推荐结果是否令人满意。
#


# # 保存特征
#
#
训练完模型后，我们得到每个用户、电影对应的特征向量，接下来将这些特征向量保存到本地，这样在进行推荐时，不需要使用神经网络重新提取特征，节省时间成本。
#
# 保存特征的流程是：
# - 加载预训练好的模型参数。
# -
输入数据集的数据，提取整个数据集的用户特征和电影特征。注意数据输入到模型前，要先转成内
```

置的tensor类型并保证尺寸正确。

- 分别得到用户特征向量和电影特征向量，使用Pickle库保存字典形式的特征向量。

#

#

使用用户和电影ID为索引，以字典格式存储数据，可以通过用户或者电影的ID索引到用户特征和电影特征。

#

#

下面代码中，我们使用了一个Pickle库。Pickle库为python提供了一个简单的持久化功能，可以很容易的将Python对象保存到本地，但缺点是保存的文件可读性较差。

```
# In[ ]:


from PIL import Image
# 加载第三方库Pickle，用来保存Python数据到本地
import pickle
# 定义特征保存函数
def get_usr_mov_features(model, params_file_path, poster_path):
    paddle.set_device('cpu')
    usr_pkl = {}
    mov_pkl = {}

    # 定义将list中每个元素转成tensor的函数
    def list2tensor(inputs, shape):
        inputs = np.reshape(np.array(inputs).astype(np.int64), shape)
        return paddle.to_tensor(inputs)

    # 加载模型参数到模型中，设置为验证模式eval（）
    model_state_dict = paddle.load(params_file_path)
    model.load_dict(model_state_dict)
    model.eval()
    # 获得整个数据集的数据
    dataset = model.Dataset.dataset

    for i in range(len(dataset)):
        # 获得用户数据，电影数据，评分数据
        #
        本案例只转换所有在样本中出现过的user和movie，实际中可以使用业务系统中的全量数据
        usr_info, mov_info, score = dataset[i]['usr_info'], dataset[i]['mov_info'],
        dataset[i]['scores']
        usrid = str(usr_info['usr_id'])
        movid = str(mov_info['mov_id'])

        # 获得用户数据，计算得到用户特征，保存在usr_pkl字典中
        if usrid not in usr_pkl.keys():
            usr_id_v = list2tensor(usr_info['usr_id'], [1])
            usr_age_v = list2tensor(usr_info['age'], [1])
            usr_gender_v = list2tensor(usr_info['gender'], [1])
            usr_job_v = list2tensor(usr_info['job'], [1])

            usr_in = [usr_id_v, usr_gender_v, usr_age_v, usr_job_v]
            usr_feat = model.get_usr_feat(usr_in)

            usr_pkl[usrid] = usr_feat.numpy()

        # 获得电影数据，计算得到电影特征，保存在mov_pkl字典中
        if movid not in mov_pkl.keys():
            mov_id_v = list2tensor(mov_info['mov_id'], [1])
            mov_tit_v = list2tensor(mov_info['title'], [1, 1, 15])
            mov_cat_v = list2tensor(mov_info['category'], [1, 6])

            mov_in = [mov_id_v, mov_cat_v, mov_tit_v, None]
            mov_feat = model.get_mov_feat(mov_in)

            mov_pkl[movid] = mov_feat.numpy()



    print(len(mov_pkl.keys()))
    # 保存特征到本地
```

```
        pickle.dump(usr_pkl, open('./usr_feat.pkl', 'wb'))
        pickle.dump(mov_pkl, open('./mov_feat.pkl', 'wb'))
        print("usr / mov features saved!!!")


param_path = "./checkpoint/epoch9.pdparams"
poster_path = "./work/ml-1m/posters/"
get_usr_mov_features(model, param_path, poster_path)


#
保存好有效代表用户和电影的特征向量后，在下一节我们讨论如何基于这两个向量构建推荐系统。

# ## 作业 10-2
#
# 1.
以上算法使用了用户与电影的所有特征（除Poster外），可以设计对比实验，验证哪些特征是重要的，把最终的特征挑选出来。为了验证哪些特征起到关键作用，
读者可以启用或弃用其中某些特征，或者加入电影海报特征，观察是否对模型Loss或评价指标有提升。
# 1.
加入电影海报数据，验证电影海报特征（Poster）对推荐结果的影响，实现并分析推荐结果（有没有效果？为什么？）。


# 综合 py
#!/usr/bin/env python
# coding: utf-8

# 训练并保存好模型，我们可以开始实践电影推荐了，推荐方式可以有多种，比如：
# 1. 根据一个电影推荐其相似的电影。
# 2. 根据用户的喜好，推荐其可能喜欢的电影。
# 3. 给指定用户推荐与其喜好相似的用户喜欢的电影。
#
#
# 这里我们实现第二种推荐方式，另外两种留作实践作业。
#
# # 根据用户喜好推荐电影
#
#
在前面章节，我们已经完成了神经网络的设计，并根据用户对电影的喜好（评分高低）作为训练指标完成训练。神经网络有两个输入，用户数据和电影数据，通过神经网络提取用户特征和电影特征，并计算特征之间的相似度，相似度的大小和用户对该电影的评分存在对应关系。即如果用户对这个电影感兴趣，那么对这个电影的评分也是偏高的，最终神经网络输出的相似度就更大一些。完成训练后，我们就可以开始给用户推荐电影了。
#
#
根据用户喜好推荐电影，是通过计算用户特征和电影特征之间的相似性，并排序选取相似度最大的结果来进行推荐，流程如下：
#
#
# <center><img src="
https://ai-studio-static-online.cdn.bcebos.com/368f768dea324a46a15519c731434515701cc1b0bad642949c22d90d71efc349" width="800" ></center>
#
#
# 从计算相似度到完成推荐的过程，步骤包括：
#
# 1. 读取保存的特征，根据一个给定的用户ID、电影ID，我们可以索引到对应的特征向量。
# 2. 通过计算用户特征和其他电影特征向量的相似度，构建相似度矩阵。
# 3.
对这些相似度排序后，选取相似度最大的几个特征向量，找到对应的电影ID，即得到推荐清单。
# 4.
加入随机选择因素，从相似度最大的top_k结果中随机选取pick_num个推荐结果，其中pick_num必须小于top_k。
#
#

# ## 1. 读取特征向量

#
上一节我们已经训练好模型，并保存了电影特征，因此可以不用经过计算特征的步骤，直接读取特
```

```
征。
#
特征以字典的形式保存，字典的键值是用户或者电影的ID，字典的元素是该用户或电影的特征向量
。
#
# 下面实现根据指定的用户ID和电影ID，索引到对应的特征向量。

# In[1]:


get_ipython().system(' unzip -o data/data19736/ml-1m.zip -d /home/aistudio/work/')
# ! unzip -o data/data20452/save_feat.zip -d /home/aistudio/
get_ipython().system(' unzip -o data/data20452/save_feature_v1.zip -d
/home/aistudio/')


# In[2]:


import pickle
import numpy as np

mov_feat_dir = 'mov_feat.pkl'
usr_feat_dir = 'usr_feat.pkl'

usr_feats = pickle.load(open(usr_feat_dir, 'rb'))
mov_feats = pickle.load(open(mov_feat_dir, 'rb'))

usr_id = 2
usr_feat = usr_feats[str(usr_id)]

mov_id = 1
# 通过电影ID索引到电影特征
mov_feat = mov_feats[str(mov_id)]

# 电影特征的路径
movie_data_path = "./work/ml-1m/movies.dat"
mov_info = {}
# 打开电影数据文件，根据电影ID索引到电影信息
with open(movie_data_path, 'r', encoding="ISO-8859-1") as f:
    data = f.readlines()
    for item in data:
        item = item.strip().split("::")
        mov_info[str(item[0])] = item

usr_file = "./work/ml-1m/users.dat"
usr_info = {}
# 打开文件，读取所有行到data中
with open(usr_file, 'r') as f:
    data = f.readlines()
    for item in data:
        item = item.strip().split("::")
        usr_info[str(item[0])] = item

print("当前的用户是：")
print("usr_id:", usr_id, usr_info[str(usr_id)])
print("对应的特征是：", usr_feats[str(usr_id)])

print("\n当前电影是：")
print("mov_id:", mov_id, mov_info[str(mov_id)])
print("对应的特征是：")
print(mov_feat)


# 以上代码中，我们索引到 usr_id = 2 的用户特征向量，以及 mov_id = 1 的电影特征向量。

# ## 2. 计算用户和所有电影的相似度，构建相似度矩阵
#
# 如下示例均以向 userid = 2
的用户推荐电影为例。与训练一致，以余弦相似度作为相似度衡量。

# In[3]:
```

```
import paddle

# 根据用户ID获得该用户的特征
usr_ID = 2
# 读取保存的用户特征
usr_feat_dir = 'usr_feat.pkl'
usr_feats = pickle.load(open(usr_feat_dir, 'rb'))
# 根据用户ID索引到该用户的特征
usr_ID_feat = usr_feats[str(usr_ID)]

# 记录计算的相似度
cos_sims = []
# 记录下与用户特征计算相似的电影顺序

# 索引电影特征，计算和输入用户ID的特征的相似度
for idx, key in enumerate(mov_feats.keys()):
    mov_feat = mov_feats[key]
    usr_feat = paddle.to_tensor(usr_ID_feat)
    mov_feat = paddle.to_tensor(mov_feat)

    # 计算余弦相似度
    sim = paddle.nn.functional.common.cosine_similarity(usr_feat, mov_feat)
    # 打印特征和相似度的形状
    if idx==0:
        print("电影特征形状：{}，用户特征形状：{}，
        相似度结果形状：{}，相似度结果：{}".format(mov_feat.shape, usr_feat.shape,
        sim.numpy().shape, sim.numpy()))
    #
    从形状为（1，1）的相似度sim中获得相似度值sim.numpy()[0]，并添加到相似度列表cos_sim
    s中
    cos_sims.append(sim.numpy()[0])


# ## 3. 对相似度排序，选出最大相似度
#
#
使用np.argsort()函数完成从小到大的排序，注意返回值是原列表位置下标的数组。因为cos_sims
 和
mov_feats.keys()的顺序一致，所以都可以用index数组的内容索引，获取最大的相似度值和对应
电影。
#
# 处理流程是先计算相似度列表
cos_sims，将其排序后返回对应的下标列表index，最后从cos_sims和mov_info中取出相似度值和
对应的电影信息。
#
#
这个处理流程只是展示推荐系统的推荐效果，实际中推荐系统需要采用效率更高的工程化方案，建
立"召回+排序"的检索系统。这些检索系统的架构才能应对推荐系统对大量线上需求的实时响应。

# In[4]:


# 对相似度排序，获得最大相似度在cos_sims中的位置
index = np.argsort(cos_sims)
# 打印相似度最大的前topk个位置
topk = 5
print("相似度最大的前{}个索引是{}\n对应的相似度是：{}\n".format(topk, index[-topk:],
[cos_sims[k] for k in index[-topk:]]))

for i in index[-topk:]:
    print("对应的电影分别是：movie:{}".format(mov_info[list(mov_feats.keys())[i]]))


# 以上结果可以看出，给用户推荐的电影多是Drama、War、Thriller类型的电影。
#
# 是不是到这里就可以把结果推荐给用户了？还有一个小步骤我们继续往下看。

# ## 4.加入随机选择因素，使得每次推荐的结果有"新鲜感"
#
#
```

为了确保推荐的多样性，维持用户阅读推荐内容的“新鲜感”，每次推荐的结果需要有所不同，我们随机抽取top_k结果中的一部分，作为给用户的推荐。比如从相似度排序中获取10个结果，每次随机抽取6个结果推荐给用户。

#

#

使用np.random.choice函数实现随机从top_k中选择一个未被选的电影，不断选择直到选择列表res长度达到pick_num为止，其中pick_num必须小于top_k。

#

读者可以反复运行本段代码，观测推荐结果是否有所变化。

#

代码实现如下：

```
# In[5]:


top_k, pick_num = 10, 6

# 对相似度排序，获得最大相似度在cos_sims中的位置
index = np.argsort(cos_sims)[-top_k:]

print("当前的用户是：")
# usr_id, usr_info 是前面定义、读取的用户ID、用户信息
print("usr_id:", usr_id, usr_info[str(usr_id)])
print("推荐可能喜欢的电影是：")
res = []

# 加入随机选择因素，确保每次推荐的结果稍有差别
while len(res) < pick_num:
    val = np.random.choice(len(index), 1)[0]
    idx = index[val]
    mov_id = list(mov_feats.keys())[idx]
    if mov_id not in res:
        res.append(mov_id)

for id in res:
    print("mov_id:", id, mov_info[str(id)])


# 最后，我们将根据用户ID推荐电影的实现封装成一个函数，方便直接调用，其函数实现如下。

# In[6]:


# 定义根据用户兴趣推荐电影
def recommend_mov_for_usr(usr_id, top_k, pick_num, usr_feat_dir, mov_feat_dir,
mov_info_path):
    assert pick_num <= top_k
    # 读取电影和用户的特征
    usr_feats = pickle.load(open(usr_feat_dir, 'rb'))
    mov_feats = pickle.load(open(mov_feat_dir, 'rb'))
    usr_feat = usr_feats[str(usr_id)]

    cos_sims = []

    # with dygraph.guard():
    paddle.disable_static()
    # 索引电影特征，计算和输入用户ID的特征的相似度
    for idx, key in enumerate(mov_feats.keys()):
        mov_feat = mov_feats[key]
        usr_feat = paddle.to_tensor(usr_feat)
        mov_feat = paddle.to_tensor(mov_feat)
        # 计算余弦相似度
        sim = paddle.nn.functional.common.cosine_similarity(usr_feat, mov_feat)

        cos_sims.append(sim.numpy()[0])
    # 对相似度排序
    index = np.argsort(cos_sims)[-top_k:]

    mov_info = {}
    # 读取电影文件里的数据，根据电影ID索引到电影信息
    with open(mov_info_path, 'r', encoding="ISO-8859-1") as f:
        data = f.readlines()
```

```
            for item in data:
                item = item.strip().split("::")
                mov_info[str(item[0])] = item

        print("当前的用户是：")
        print("usr_id:", usr_id)
        print("推荐可能喜欢的电影是：")
        res = []

        # 加入随机选择因素，确保每次推荐的都不一样
        while len(res) < pick_num:
            val = np.random.choice(len(index), 1)[0]
            idx = index[val]
            mov_id = list(mov_feats.keys())[idx]
            if mov_id not in res:
                res.append(mov_id)

        for id in res:
            print("mov_id:", id, mov_info[str(id)])


# In[7]:


movie_data_path = "./work/ml-1m/movies.dat"
top_k, pick_num = 10, 6
usr_id = 2
recommend_mov_for_usr(usr_id, top_k, pick_num, 'usr_feat.pkl', 'mov_feat.pkl',
movie_data_path)


#
从上面的推荐结果来看，给ID为2的用户推荐的电影多是Drama、War类型的。我们可以通过用户的ID从已知的评分数据中找到其评分最高的电影，观察和推荐结果的区别。
#
#
下面代码实现给定用户ID，输出其评分最高的topk个电影信息，通过对比用户评分最高的电影和当前推荐的电影结果，观察推荐是否有效。

# In[8]:


# 给定一个用户ID，找到评分最高的topk个电影

usr_a = 2
topk = 10

########################################
## 获得ID为usr_a的用户评分过的电影及对应评分 ##
########################################
rating_path = "./work/ml-1m/ratings.dat"
# 打开文件，ratings_data
with open(rating_path, 'r') as f:
    ratings_data = f.readlines()

usr_rating_info = {}
for item in ratings_data:
    item = item.strip().split("::")
    # 处理每行数据，分别得到用户ID，电影ID，和评分
    usr_id,movie_id,score = item[0],item[1],item[2]
    if usr_id == str(usr_a):
        usr_rating_info[movie_id] = float(score)

# 获得评分过的电影ID
movie_ids = list(usr_rating_info.keys())
print("ID为 {} 的用户，评分过的电影数量是: ".format(usr_a), len(movie_ids))

####################################
## 选出ID为usr_a评分最高的前topk个电影 ##
####################################
ratings_topk = sorted(usr_rating_info.items(), key=lambda item:item[1])[-topk:]
```

```
movie_info_path = "./work/ml-1m/movies.dat"
# 打开文件，编码方式选择ISO-8859-1，读取所有数据到data中
with open(movie_info_path, 'r', encoding="ISO-8859-1") as f:
    data = f.readlines()

movie_info = {}
for item in data:
    item = item.strip().split("::")
    # 获得电影的ID信息
    v_id = item[0]
    movie_info[v_id] = item

for k, score in ratings_topk:
    print("电影ID: {}, 评分是: {}, 电影信息: {}".format(k, score, movie_info[k]))


#
通过上述代码的输出可以发现，Drama类型的电影是用户喜欢的类型，可见推荐结果和用户喜欢的电影类型是匹配的。但是推荐结果仍有一些不足的地方，这些可以通过改进神经网络模型等方式来进一步调优。

# # 从推荐案例的三点思考
#
# 1. Deep Learning is all about “Embedding Everything”。不难发现，深度学习建模是套路满满的。任何事物均用向量的方式表示，可以直接基于向量完成“分类”或“回归”任务；也可以计算多个向量之间的关系，无论这种关系是“相似性”还是“比较排序”。在深度学习兴起不久的2015年，当时AI相关的国际学术会议上，大部分论文均是将某个事物Embedding后再进行挖掘，火热的程度仿佛即使是路边一块石头，也要Embedding一下看看是否能挖掘出价值。直到近些年，能够Embedding的事物基本都发表过论文，Embeddding的方法也变得成熟，这方面的论文才逐渐有减少的趋势。
#
# 2.
在深度学习兴起之前，不同领域之间的迁移学习往往要用到很多特殊设计的算法。但深度学习兴起后，迁移学习变得尤其自然。训练模型和使用模型未必是同样的方式，中间基于Embedding的向量表示，即可实现不同任务交换信息。例如本章的推荐模型使用用户对电影的评分数据进行监督训练，训练好的特征向量可以用于计算用户与用户的相似度，以及电影与电影之间的相似度。对特征向量的使用可以极其灵活，而不局限于训练时的任务。
#
# 3.
网络调参：神经网络模型并没有一套理论上可推导的最优规则，实际中的网络设计往往是在理论和经验指导下的“探索”活动。例如推荐模型的每层网络尺寸的设计遵从了信息熵的原则，原始信息量越大对应表示的向量长度就越长。但具体每一层的向量应该有多长，往往是根据实际训练的效果进行调整。所以，建模工程师被称为数据处理工程师和调参工程师是有道理的，大量的精力花费在处理样本数据和模型调参上。
#
# <center><img src="
https://ai-studio-static-online.cdn.bcebos.com/89cd810fd1034864b55e52465424ff763dcfc8efe86d492d9a58edf536d9393e" width="1000" ></center>
#
# <br>
#

# # 在工业实践中的推荐系统
#
#
本章介绍了比较简单的推荐系统构建方法，在实际应用中，验证一个推荐系统的好坏，除了预测准确度，还需要考虑多方面的因素，比如多样性、新颖性，甚至商业目标匹配度等。要实践一个好的推荐系统，值得更深入的探索研究。下面将工业实践推荐系统还需要考虑的主要问题做一个概要性的介绍。
#
#
1.  **推荐来源**：推荐来源会更加多样化，除了使用深度学习模型的方式，还大量使用标签匹配的个性化推荐方式。此外，推荐热门的内容，具有时效性的内容和一定探索性的内容，都非常关键。对于新闻类的内容推荐，用户不希望地球人都在谈论的大事自己毫无所知，期望更快更全面的了解。如果用户经常使用的推荐产品总推荐“老三样”，会使得用户丧失“新鲜感”而流失。因此，除了推荐一些用户喜欢的内容之外，谨慎的推荐一些用户没表达过喜欢的内容，可探索用户更广泛的兴趣领域，以便有更多不重复的内容可以向用户推荐。
#
#
2.  **检索系统**：将推荐系统构建成“召回+排序”架构的高性能检索系统，以更短的特征向量建倒排索引。在“召回＋排序”的架构下，通常会训练出两种不同长度的特征向量，使用较短的特征向量做召回系统，从海量候选中筛选出几十个可能候选。使用较短的向量做召回，性能高但不够准确
```

，然后使用较长的特征向量做几十个候选的精细排序，因为待排序的候选很少，所以性能低一些也影响不大。
#
#
3. **冷启动问题**：现实中推荐系统往往要在产品运营的初期一起上线，但这时候系统尚没有用户行为数据的积累。这时，我们往往建立一套专家经验的规则系统，比如一个在美妆行业工作的店小二对各类女性化妆品偏好是非常了解的。通过规则系统运行一段时间积累数据后，再逐渐转向机器学习的系统。很多推荐系统也会主动向用户收集一些信息，比如大家注册一些资讯类APP时，经常会要求选择一些兴趣标签。
#
#
4. **推荐系统的评估**：推荐系统的评估不仅是计算模型Loss所能代表的，是使用推荐系统用户的综合体验。除了采用更多代表不同体验的评估指标外（准确率、召回率、覆盖率、多样性等），还会从两个方面收集数据做分析：
#
（1）行为日志：如用户对推荐内容的点击率，阅读市场，发表评论，甚至消费行为等。
#
#
（2）人工评估：选取不同的具有代表性的评估员，从兴趣相关度、内容质量、多样性、时效性等多个维度评估。如果评估员就是用户，通常是以问卷调研的方式下发和收集。
#
#
其中，多样性的指标是针对探索性目标的。而推荐的覆盖度也很重要，代表了所有的内容有多少能够被推荐系统送到用户面前。如果推荐每次只集中在少量的内容，大部分内容无法获得用户流量的话，会影响系统内容生态的健康。比如电商平台如果只推荐少量大商家的产品给用户，多数小商家无法获得购物流量，会导致平台上的商家集中度越来越高，生态不再繁荣稳定。
#
#
从上述几点可见，搭建一套实用的推荐系统，不只是一个有效的推荐模型。要从业务的需求场景出发，构建完整的推荐系统，最后再实现模型的部分。如果技术人员的视野只局限于模型本身，是无法在工业实践中搭建一套有业务价值的推荐系统的。
#
<img src="
https://ai-studio-static-online.cdn.bcebos.com/2e9d3c764d764f78ab10efcbfbfd018c78f334c244314ec6a23751af0b74792d" width="600" >
#
图3：推荐系统的全流程

#

作业
#
1、设计并完成两个推荐系统，根据相似用户推荐电影（user-based）和
根据相似电影推荐电影（item-based），并分析三个推荐系统的推荐结果差异。
#
#
上文中，我们已经将映射后的用户特征和电影特征向量保存在了本地，通过两者的相似度计算结果进行推荐。实际上，我们还可以计算用户之间的相似度矩阵和电影之间的相似度矩阵，实现根据相似用户推荐电影和根据相似电影推荐电影。
#
#
#
2、构建一个【热门】、【新品】和【个性化推荐】三条推荐路径的混合系统。构建更贴近真实场景的推荐系统，而不仅是个性化推荐模型，每次推荐10条，三种各占比例2、3、5条，每次的推荐结果不同。
#
3、推荐系统的案例，实现本地的版本（非AI
Studio上实现），进行训练和预测并截图提交。有助于大家掌握脱离AI
Studio平台，使用本地机器完成建模的能力。