

dl-re-movies

大数据班 软工1608班 20165277 赵煜

项目简介：

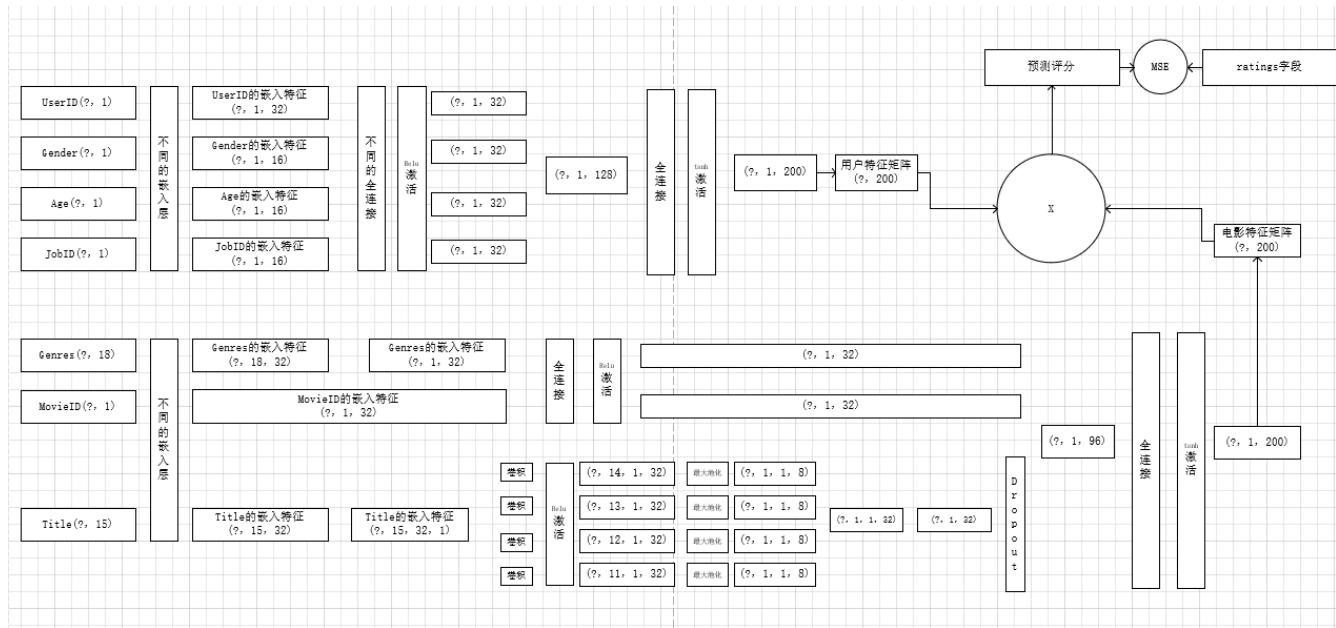
- dl_re_web: Web 项目的文件夹
- re_sys: Web app
 - model: 百度云下载之后，把model放到该文件夹下
 - [recommend]: 网络模型相关
 - data: 训练数据集
 - DataSet.py
 - re_model.py: 网络模型类
 - utils.py: 工具、爬虫
- static: Web 页面静态资源
- templates: 为 Web 页面的 Html 页面
- venv: Django 项目资源文件夹
- db.sqlite3: Django 自带的数据库
- manage.py: Django 执行脚本

模型百度云链接: <https://pan.baidu.com/s/1y03hHrEZio57xUqh33SjtA> 提取码: 6xpt

- 项目背景

本系统将神经网络在自然语言处理与电影推荐相结合，利用MovieLens数据集训练一个基于文本的卷积神经网络，实现电影个性化推荐系统。最后使用django框架并结合豆瓣爬虫，搭建推荐系统web端服务。

- 主要实现功能
 - 给用户推荐喜欢的电影
 - 推荐相似的电影
 - 推荐看过的用户还喜欢看的电影
- 网络模型



一. 数据处理

1. MovieLens数据集

- 用户数据users.dat

性别字段：将‘F’和‘M’转换为0和1

年龄字段：转为连续数字

- 电影数据movies.dat

流派字段：部分电影不仅只有一个分类，所以将该字段转为数字列表

标题字段：同上，创建英文标题的数字字典，并生成数字列表，并去掉标题中的年份

注：为方便网络处理，以上两字段长度需要统一

- 评分数据ratings.dat

数据处理完之后将三个表做 inner merge，并保存为模型文件 data_preprocess.pkl

2. 处理后的数据

	UserID	MovieID	ratings	Gender	Age	JobID	Title	Genres
0	1	1193	5	0	0	10	[3596, 3731, 1517, 2824, 308, 4219, 916, 916, ...]	[14, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 1...
1	2	1193	5	1	5	16	[3596, 3731, 1517, 2824, 308, 4219, 916, 916, ...]	[14, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 1...
2	12	1193	4	1	6	12	[3596, 3731, 1517, 2824, 308, 4219, 916, 916, ...]	[14, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 1...
3	15	1193	4	1	6	7	[3596, 3731, 1517, 2824, 308, 4219, 916, 916, ...]	[14, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 1...
4	17	1193	5	1	3	1	[3596, 3731, 1517, 2824, 308, 4219, 916, 916, ...]	[14, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 1...

我们看到部分字段是类型性变量，如 UserID、MovieID 这样非常稀疏的变量，如果使用 one-hot，那么数据的维度会急剧膨胀，算法的效率也会大打折扣。

二. 建模&训练

针对处理后数据的不同字段进行模型的搭建

1. 嵌入层

根据上文，为了解决数据稀疏问题，one-hot 的矩阵相乘可以简化为查表操作，这大大降低了运算量。我们不是每一个词用一个向量来代替，而是替换为用于查找嵌入矩阵中向量的索引，在网络的训练过程中，嵌入向量也会更新，我们也可以探索在高维空间中词语之间的相似性。

本系统使用tensorflow的 `tf.nn.embedding_lookup` ,就是根据input_ids中的id, 寻找embeddings中的第id行。比如 `input_ids=[1,3,5]`, 则找出embeddings中第1, 3, 5行, 组成一个tensor返回。`tf.nn.embedding_lookup` 不是简单的查表, id对应的向量是可以训练的, 训练参数个数应该是 `category num*embedding size`, 也可以说lookup是一种全连接层。

- 解析：

1. 创建嵌入矩阵，我们要决定每一个索引需要分配多少个潜在因子，这大体上意味着我们想要多长的向量，通常使用的情况是长度分配为32和50，此处选择32和16，所以我们看到各字段嵌入矩阵的shape第1个维度，也就是第2个数字要么为32，要么为16；
 2. 而嵌入矩阵第0个纬度为6041、2、7、21，也就是嵌入矩阵的行数，也就代表着这四个字段unique值有多少个，例如Gender的值只有0和1（经过数据处理）其嵌入矩阵就有2行
 3. 到现在，想必大家可以清楚嵌入矩阵的好处了，我们以userId字段为例，使用one-hot编码，数据就需要增加`数据量x6041`个数据，如果数据量较大，或者字段的unique值较多，在训练时则会耗费大量资源，但是如果使用嵌入矩阵，我们仅仅只用创建一个`6041x32`的矩阵，然后使用`tf.nn.embedding_lookup`与UserID字段的数据进行全连接（相当于查表操作），即可用一个一维的长度为32的数组表示出该UserID，大大简化了运算的耗时。
 4. 在上一点已经讲过使用`tf.nn.embedding_lookup`与UserID字段的数据进行全连接（相当于查表操作），则每个嵌入层的shape应该是这样的`(数据量, 字段长度, 索引长度)`，数据量可以设计为每个epoch的大小；对于User数据来说，字段长度都为1，因为用一个值就能表示改独一无二的值，如果对于文本，则可能需要使用数组来表示，即字段长度可能大于1，稍后会在Movie数据处理中进一步解释；索引长度则是嵌入矩阵的潜在因子。

例子：对数据集字段 UserID、Gender、Age、JobID 分别构建嵌入矩阵和嵌入层

```
1 def create_user_embedding(self, uid, user_gender, user_age, user_job):
2     with tf.name_scope("user_embedding"):
3         uid_embed_matrix = tf.Variable(tf.random_uniform([self.uid_max, self.embed_dim], -1, 1),
4                                         name="uid_embed_matrix") # (6041,32)
5         uid_embed_layer = tf.nn.embedding_lookup(uid_embed_matrix, uid, name="uid_embed_layer") #
6         # (?,1,32)
7
8         gender_embed_matrix = tf.Variable(tf.random_uniform([self.gender_max, self.embed_dim // 2],
9                                         -1, 1),
10                                         name="gender_embed_matrix") # (2,16)
11         gender_embed_layer = tf.nn.embedding_lookup(gender_embed_matrix, user_gender,
12                                         name="gender_embed_layer") # (?,1,16)
13
14         age_embed_matrix = tf.Variable(tf.random_uniform([self.age_max, self.embed_dim // 2], -1,
15                                         1),
16                                         name="age_embed_matrix") # (7,16)
17         age_embed_layer = tf.nn.embedding_lookup(age_embed_matrix, user_age,
18                                         name="age_embed_layer")# (?,1,16)
19
20         job_embed_matrix = tf.Variable(tf.random_uniform([self.job_max, self.embed_dim // 2], -1,
21                                         1),
```

```

17                               name="job_embed_matrix") # (21,16)
18   job_embed_layer = tf.nn.embedding_lookup(job_embed_matrix, user_job,
19   name="job_embed_layer")# (?,1,16)
19   return uid_embed_layer, gender_embed_layer, age_embed_layer, job_embed_layer

```

类似地，我们在相应代码中分别创建了电影数据的MovieID、Genres、Title的嵌入矩阵，其中需要特别注意的是：

1. Title嵌入层的shape是 `(?, 15, 32)`，“?”代表了一个epoch的数量，32代表了自定义选择的潜在因子数量，15则代表了该字段的每一个unique值都需要一个长度为15的向量来表示。
2. Genres嵌入层的shape是 `(?, 1, 32)`，由于一个电影的Genres（电影的类型），可能属于多个类别，所以该字段的需要做特殊的处理，即把第1纬度上的向量进行加和，这样做其实削减了特征的表现，但是又防止比如仅仅只推荐相关类型的电影。

- 综上，经过嵌入层，我们得到一下模型：

针对User数据

模型名称	shape
uid_embed_matrix	(6041, 32)
gender_embed_matrix	(2, 16)
age_embed_matrix	(7, 16)
job_embed_matrix	(21, 16)
uid_embed_layer	(?, 1, 32)
gender_embed_layer	(?, 1, 16)
age_embed_layer	(?, 1, 16)
job_embed_layer	(?, 1, 16)

针对Movie数据

模型名称	shape
movie_id_embed_matrix	(3953, 32)
movie_categories_embed_matrix	(19, 32)
movie_title_embed_matrix	(5215, 32)
movie_id_embed_layer	(?, 1, 32)
movie_categories_embed_layer	(?, 1, 32)
movie_title_embed_layer	(?, 15, 32)

2. 文本卷积层

本文仅介绍了推导过程，并为介绍卷积层设计的思路。设计思路请看 [参考文献](#)

文本卷积层仅涉及到电影数据的Title字段，其实Genres字段也是可以进行文本卷积设计的，但是上文解释过，考虑到推荐数据字段的影响，对Genres仅设计了常规的网络。

卷积过程涉及到一下几个参数：

name&value	解释
windows_size=[2, 3, 4, 5]	不同卷积的滑动窗口是可变的
filter_num=8	卷积核（滤波器）的数量
filter_weight = (windows_size, 32, 1, filter_num)	卷积核的权重，四个参数分别为（高度，宽度，输入通道数，输出通道数）
filter_bias=8	卷积核的偏置=卷积核的输出通道数=卷积核的数量

- 过程

我们将Title字段潜入层的输出 `movie_title_embed_layer` (`shape=(?, 15, 32)`)，作为卷积层的输入，所以我们先把 `movie_title_embed_layer` 扩展一个维度，`shape` 变为 `(?, 15, 32, 1)`，四个参数分别为 `(batch, height, width, channels)`

```
1 | movie_title_embed_layer_expand = tf.expand_dims(movie_title_embed_layer, -1) # 在最后加上一个维度
```

使用不同尺寸的卷积核做卷积和最大池化，相关参数的变化不再赘述

```
1 | pool_layer_lst = []
2 | for window_size in self.window_sizes:
3 |     with tf.name_scope("movie_txt_conv_maxpool_{}".format(window_size)):
4 |         # 卷积核权重
5 |         filter_weights = tf.Variable(tf.truncated_normal([window_size, self.embed_dim, 1,
6 |             self.filter_num], stddev=0.1), name="filter_weights")
7 |
8 |         # 卷积核偏执
9 |         filter_bias = tf.Variable(tf.constant(0.1, shape=[self.filter_num]), name="filter_bias")
10 |
11 |         # 卷积层 第一个参数为：输入 第二个参数为：卷积核权重 第三个参数为：步长
12 |         conv_layer = tf.nn.conv2d(movie_title_embed_layer_expand, filter_weights, [1, 1, 1,
13 |             1], padding="VALID", name="conv_layer")
14 |
15 |         # 激活层 参数的shape保持不变
16 |         relu_layer = tf.nn.relu(tf.nn.bias_add(conv_layer, filter_bias), name="relu_layer")
17 |
18 |         # 池化层 第一个参数为：输入 第二个参数为：池化窗口大小 第三个参数为：步长
19 |         maxpool_layer = tf.nn.max_pool(relu_layer, [1, self.sentences_size - window_size +
20 |             1, 1, 1], [1, 1, 1, 1], padding="VALID", name="maxpool_layer")
```

可得到：

window_size	filter_weights	filter_bias	conv_layer	relu_layer	maxpool_layer
2	(2, 32, 1, 8)	8	(?, 14, 1, 8)	(?, 14, 1, 8)	(?, 1, 1, 8)
3	(3, 32, 1, 8)	8	(?, 13, 1, 8)	(?, 14, 1, 8)	(?, 1, 1, 8)
4	(4, 32, 1, 8)	8	(?, 12, 1, 8)	(?, 14, 1, 8)	(?, 1, 1, 8)
5	(5, 32, 1, 8)	8	(?, 11, 1, 8)	(?, 14, 1, 8)	(?, 1, 1, 8)

例子解析：

我们考虑window_size=2的情况，首先我们得到嵌入层输出，并对其增加一个维度得到

`movie_title_embed_layer_expand (shape=(?, 15, 32, 1))`，其作为卷积层的输入。

卷积核的参数 `filter_weights` 为(2, 32, 1, 8)，表示卷积核的高度为2，宽度为32，输入通道为1，输出通道为32。其中输出通道与上一层的输入通道相同。

卷积层在各个维度上的步长都为1，且padding的方式为VALID，则可得到卷基层的shape为 (?, 14, 1, 8)。

卷积之后使用relu函数进行激活，并且加上偏置，shape保持不变。

最大池化的窗口为 (1, 14, 1, 1)，且在每个维度上的步长都为1，即可得到池化后的shape为 (?, 1, 1, 8)。

依次类推，当window_size为其他时，也能得到池化层输出shape为 (?, 1, 1, 8)。

得到四个卷积、池化的输出之后，我们使用如下代码将池化层的输出根据第3维，也就是第四个参数相连，变形为 (?, 1, 1, 32)，再变形为三维 (?, 1, 32)。

```

1 pool_layer = tf.concat(pool_layer_lst, 3, name="pool_layer") # (?, 1, 1, 32)
2 max_num = len(self.window_sizes) * self.filter_num # 32
3 pool_layer_flat = tf.reshape(pool_layer, [-1, 1, max_num], name="pool_layer_flat") # (?, 1, 32) 其实仅仅是减少了一个纬度，? 仍然为每一批批量

```

最后为了正则化防止过拟合，经过dropout层处理，输出shape为 (?, 1, 32)。

3. 全连接层

对上文所得到的嵌入层的输出和卷基层的输出进行全连接。

- 对User数据的嵌入层进行全连接，最终得到输出特征的shape为 (?, 200)

```

1 def create_user_feature_layer(self, uid_embed_layer, gender_embed_layer, age_embed_layer,
2 job_embed_layer):
3     with tf.name_scope("user_fc"):
4         # 第一层全连接 改变最后一维
5         uid_fc_layer = tf.layers.dense(uid_embed_layer, self.embed_dim, name="uid_fc_layer",
6 activation=tf.nn.relu)
6         gender_fc_layer = tf.layers.dense(gender_embed_layer, self.embed_dim,
7 name="gender_fc_layer",
8                                         activation=tf.nn.relu)
7         age_fc_layer = tf.layers.dense(age_embed_layer, self.embed_dim, name="age_fc_layer",
8 activation=tf.nn.relu)
8         job_fc_layer = tf.layers.dense(job_embed_layer, self.embed_dim, name="job_fc_layer",
9 activation=tf.nn.relu)
9         # (?, 1, 32)
10
11     # 第二层全连接

```

```

12         user_combine_layer = tf.concat([uid_fc_layer, gender_fc_layer, age_fc_layer,
13                                         job_fc_layer], 2) # (?, 1, 128)
14         user_combine_layer = tf.contrib.layers.fully_connected(user_combine_layer, 200,
15                                         tf.tanh) # (?, 1, 200)
14         user_combine_layer_flat = tf.reshape(user_combine_layer, [-1, 200]) # (?, 200)
15     return user_combine_layer, user_combine_layer_flat

```

- 同理对Movie数据同样进行两层全连接，最终得到输出特征的shape为(?, 200)

```

1 def create_movie_feature_layer(self, movie_id_embed_layer, movie_categories_embed_layer,
2                               dropout_layer):
3     with tf.name_scope("movie_fc"):
4         # 第一层全连接
5         movie_id_fc_layer = tf.layers.dense(movie_id_embed_layer, self.embed_dim,
6                                           name="movie_id_fc_layer",
7                                           activation=tf.nn.relu) # (?, 1, 32)
8         movie_categories_fc_layer = tf.layers.dense(movie_categories_embed_layer, self.embed_dim,
9                                           name="movie_categories_fc_layer",
10                                         activation=tf.nn.relu) # (?, 1, 32)
11
12         # 第二层全连接
13         movie_combine_layer = tf.concat([movie_id_fc_layer, movie_categories_fc_layer,
14                                         dropout_layer], 2) # (?, 1, 96)
15         movie_combine_layer = tf.contrib.layers.fully_connected(movie_combine_layer, 200, tf.tanh)
15         # (?, 1, 200)
16
17         movie_combine_layer_flat = tf.reshape(movie_combine_layer, [-1, 200])
18     return movie_combine_layer, movie_combine_layer_flat

```

4. 构建计算图&训练

构建计算图，训练。问题回归为简单的将用户特征和电影特征做矩阵乘法得到一个预测评分，损失为均方误差。

```

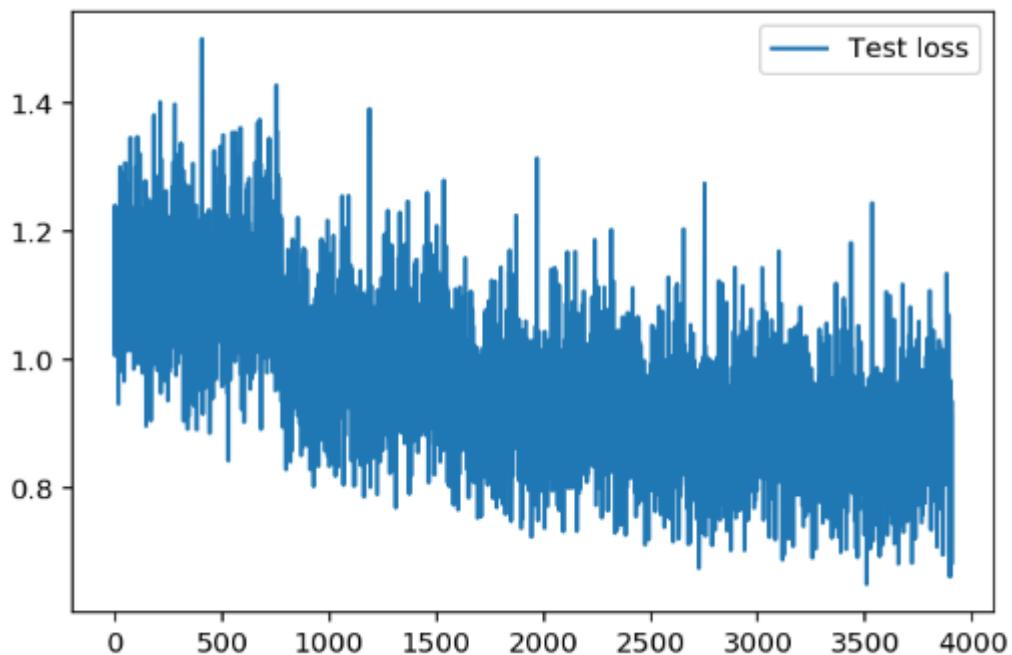
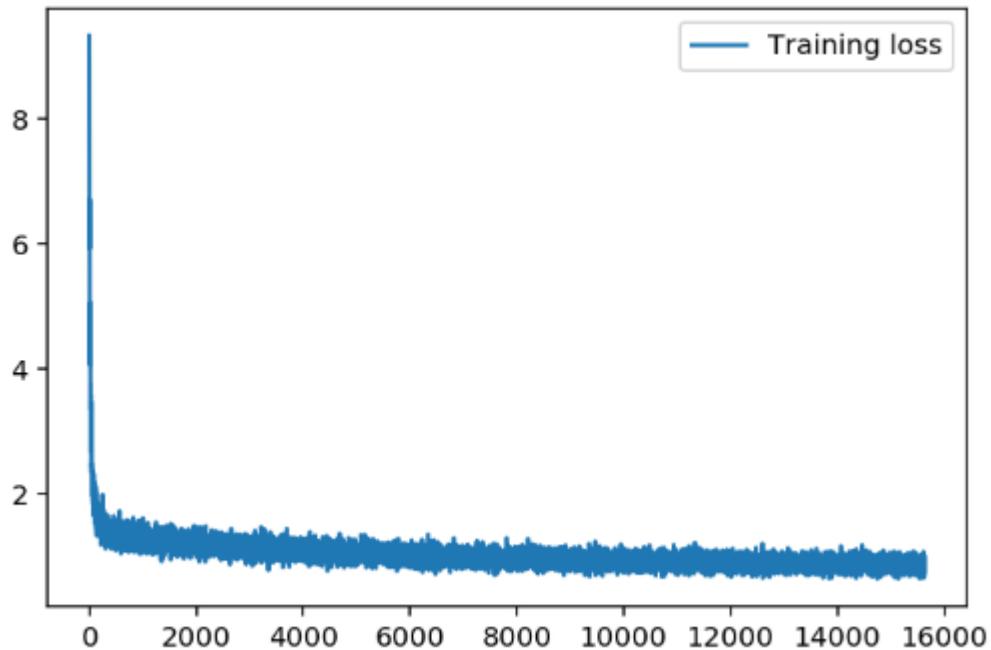
1 inference = tf.reduce_sum(user_combine_layer_flat * movie_combine_layer_flat, axis=1)
2 inference = tf.expand_dims(inference, axis=1)
3 cost = tf.losses.mean_squared_error(targets, inference)
4 loss = tf.reduce_mean(cost)
5 global_step = tf.Variable(0, name="global_step", trainable=False)
6 optimizer = tf.train.AdamOptimizer(lr) # 传入学习率
7 gradients = optimizer.compute_gradients(loss) # cost
8 train_op = optimizer.apply_gradients(gradients, global_step=global_step)

```

- 模型保存

保存的模型包括：处理后的训练数据、训练完成后的网络、用户特征矩阵、电影特征矩阵。

- 损失图像



- 经过简单的调参。`batch_size` 对 Loss 的影响较大，但是 `batch_size` 过大，损失会有比较大的抖动情况。随着学习率逐渐减小，损失会先减小后增大，所以最终确定参数还是原作者的固定参数效果较好。

5. 推荐

加入了随机因素保证对相同电影推荐时推荐结果的不一致

1. 给用户推荐喜欢的电影：使用用户特征向量与电影特征矩阵计算所有电影的评分，取评分最高的 topK 个
2. 推荐相似的电影：计算选择电影特征向量与整个电影特征矩阵的余弦相似度，取相似度最大的 topK 个
3. 推荐看过的用户还喜欢看的电影
 - 3.1 首先选出喜欢某个电影的 topK 个人，得到这几个人的用户特征向量

3.2 计算这几个人对所有电影的评分

3.3 选择每个人评分最高的电影作为推荐

三. Web展示端

1. django框架开发web

由于给定的数据集中并未有用户的其它信息，所以仅展示了“**推荐相似的电影**”和“**推荐看过的用户还喜欢看的电影**”，没有展示“**给用户推荐喜欢的电影**”这个模块，并且数据集也未有电影的中文名称、图片等数据，所以我在web项目中加了一个豆瓣的爬虫，每次推荐都请求数据，并进行相应的解析和封装。

在服务器启动的时候就加载模型，并且把tensorflow的session提前封装好，在调用相关方法时，直接传入该全局session，避免了每次请求都加载模型。

前端请求推荐的耗时大部分是爬虫请求的耗时，并且访问频率过快会被豆瓣拒绝请求一段时间。

2. 展示截图

This screenshot shows the movie recommendation system's user interface. At the top, there is a navigation bar with a logo, '电影推荐' (Movie Recommendation), '首页' (Home), and a search bar. On the right, it says 'Made by 赵煜 软工1608班 20165277'. Below the navigation, the main content area has a heading '您选择的电影' (Movie you selected). It displays a movie poster for 'VAN DAMME HELSTROM' with a rating of 6.0 and the genre '极度冒险--990'. To the right is a search bar labeled 'Search Movies' with a magnifying glass icon. The heading '电影推荐系统' (Movie Recommendation System) is centered above a grid of movie cards. The first card in the grid is for 'VAN DAMME HELSTROM' with a rating of 6.0. Below the grid, the heading '相似的电影' (Similar Movies) is displayed, followed by four movie cards: '女王神剑-2373' (rating 5.7), '大地震-2535' (rating 5.9), '悍将奇兵-1518' (rating 7.4), and '暗杀-2737' (rating 8.0).

This screenshot shows the movie recommendation system's user interface, similar to the previous one but with a different focus. It features a navigation bar at the top with a logo, '电影推荐' (Movie Recommendation), '首页' (Home), and a search bar. On the right, it says 'Made by 赵煜 软工1608班 20165277'. Below the navigation, the main content area has a heading '看过该电影的用户还喜欢' (Users who watched this movie also liked). It displays four movie cards: '星球大战-260' (rating 8.4), '高地人：复仇之旅-1275' (rating 6.8), '碧血金沙-1254' (rating 8.6), and '悬崖追击-349' (rating 7.1). To the right is a search bar labeled 'Search Movies' with a magnifying glass icon. The heading '电影推荐系统' (Movie Recommendation System) is centered above the movie cards.

- 后台推荐结果

给用户推荐喜欢的电影

您的UserID: 1401--以下是给您推荐的电影:

[3368 'Big Country, The (1958)' 'Romance|Western']
[1373 'Star Trek V: The Final Frontier (1989)' 'Action|Adventure|Sci-Fi']
[919 'Wizard of Oz, The (1939)' "Adventure|Children's|Drama|Musical"]
[2005 'Goonies, The (1985)' "Adventure|Children's|Fantasy"]
[969 'African Queen, The (1951)' 'Action|Adventure|Romance|War']

推荐相似的电影

您选择的电影是: [200 'Tie That Binds, The (1995)' 'Thriller']

相似的电影是:

[1715 'Office Killer (1997)' 'Thriller']
[2184 'Trouble with Harry, The (1955)' 'Mystery|Thriller']
[1337 'Body Snatcher, The (1945)' 'Horror']
[200 'Tie That Binds, The (1995)' 'Thriller']
[3348 'Night Visitor, The (1970)' 'Crime|Thriller']

推荐看过的用户还喜欢看的电影

您选择的电影是: [1401 'Ghosts of Mississippi (1996)' 'Drama']

喜欢看这个电影的人是: [[2569 'M' 45 0]

[3584 'M' 18 0]
[3240 'M' 18 4]
[2291 'M' 45 1]
[988 'M' 50 11]]

喜欢看这个电影的人还喜欢看:

[260 'Star Wars: Episode IV - A New Hope (1977)'
'Action|Adventure|Fantasy|Sci-Fi']
[1275 'Highlander (1986)' 'Action|Adventure']
[1254 'Treasure of the Sierra Madre, The (1948)' 'Adventure']
[349 'Clear and Present Danger (1994)' 'Action|Adventure|Thriller']
[1197 'Princess Bride, The (1987)' 'Action|Adventure|Comedy|Romance']

四. 实验项目自评与总结

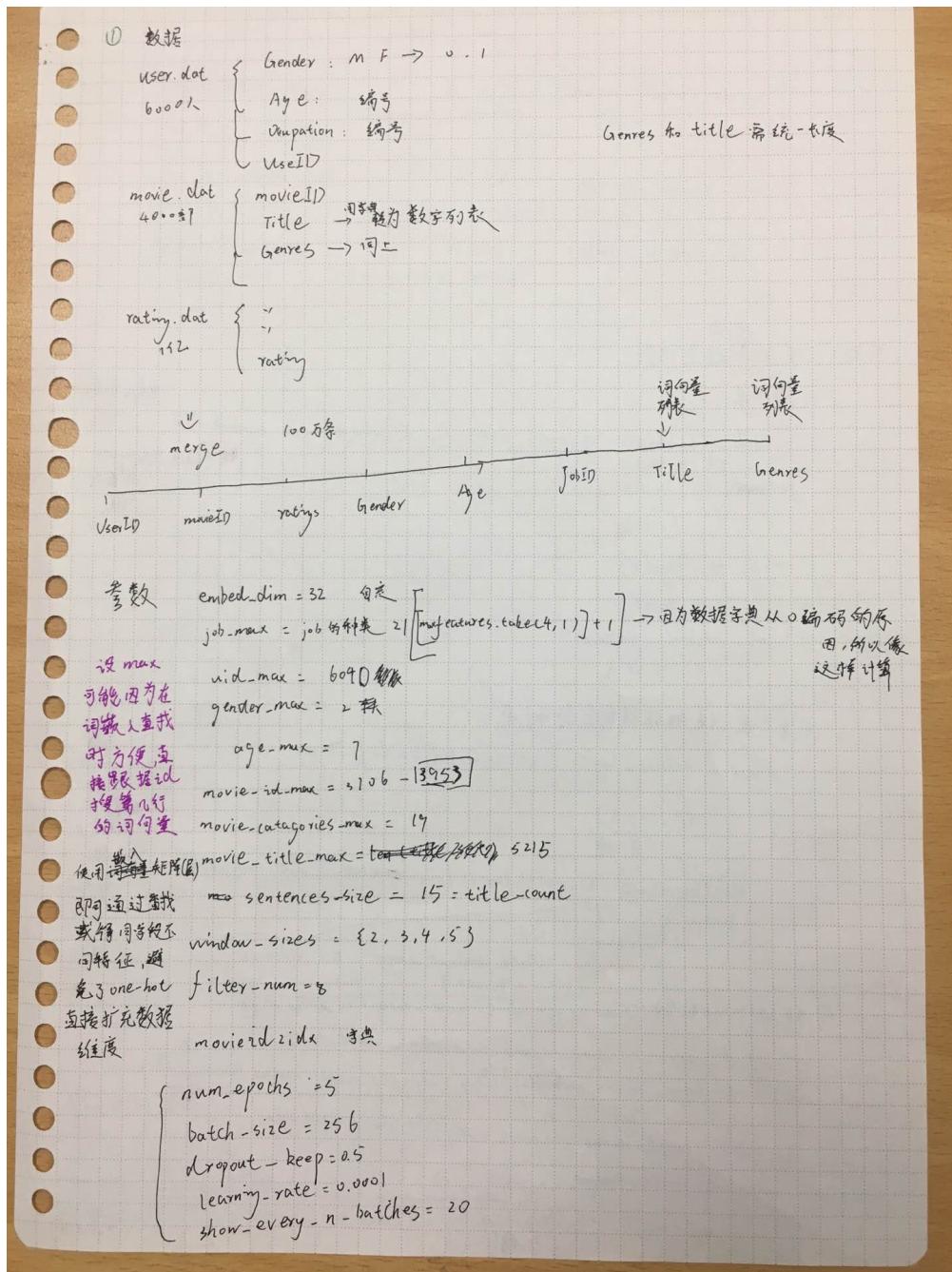
通过本次实验深度学习算是跨入了门槛，对tensorflow框架的基本使用有了一定的了解，并且此次实验的选题为推荐，是我比较喜欢的一个方向，之前对协同过滤等算法有所研究，此次利用深层网络对数据的特征进行提取更加深了我对推荐的理解。

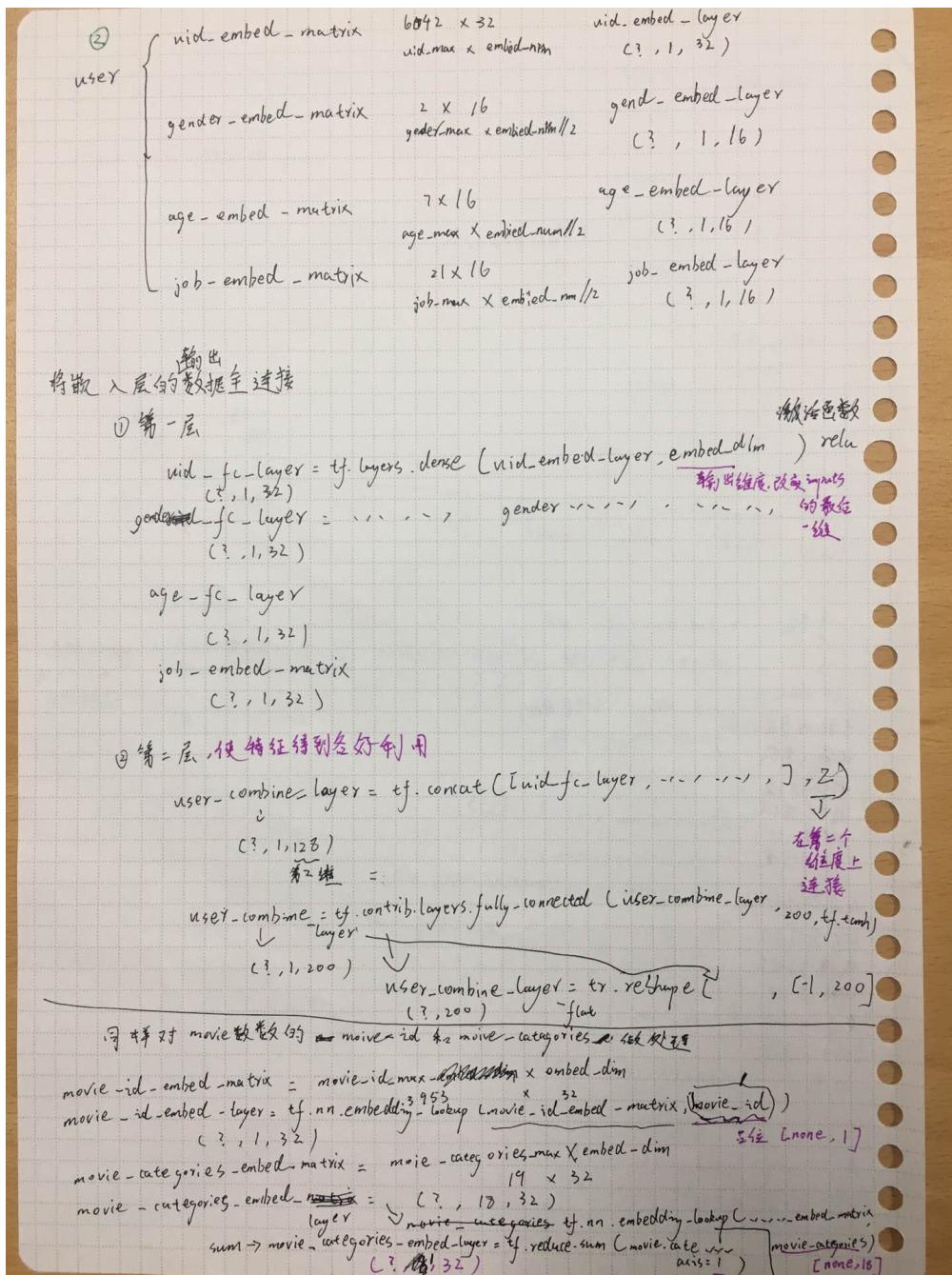
当然，本次实验的核心代码和模型架构是copy的，但我对模型的每一步都进行了演算推导，并整理成该文档，除此我把源码进行了面向对象封装，增强了源码的复用性和可用性，对推荐相关方法也进行了微小的调整，解决了模型多次加载问题，最后增加了该项目的web展示端。

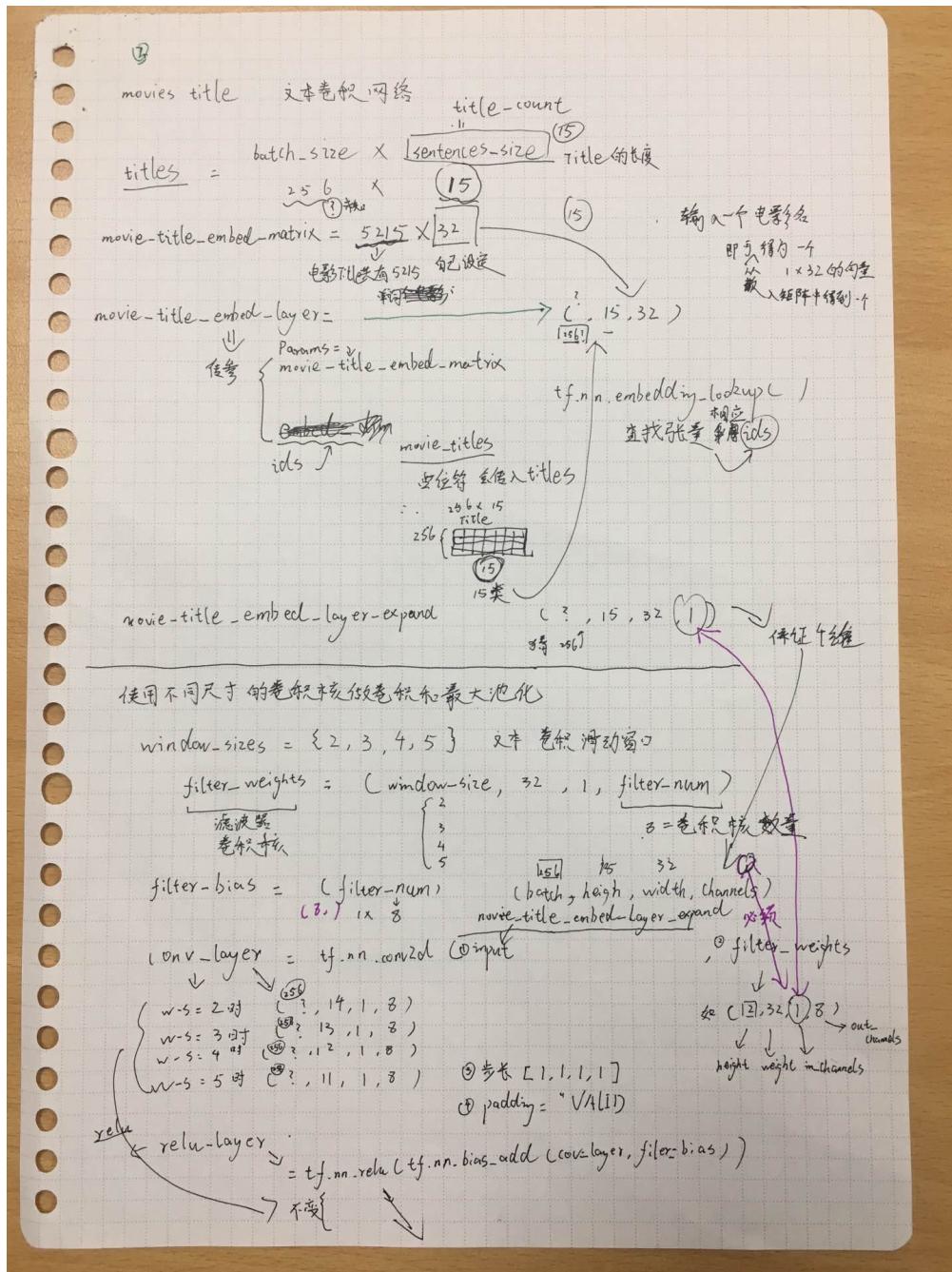
此次实验收货颇丰，但是该系统还存在一系列问题：如模型的局限性，即该系统只能对数据集中的电影和用户进行推荐，我没有再找到具有相关字段的数据，所以训练数据量相对较小，适用性也比较窄。

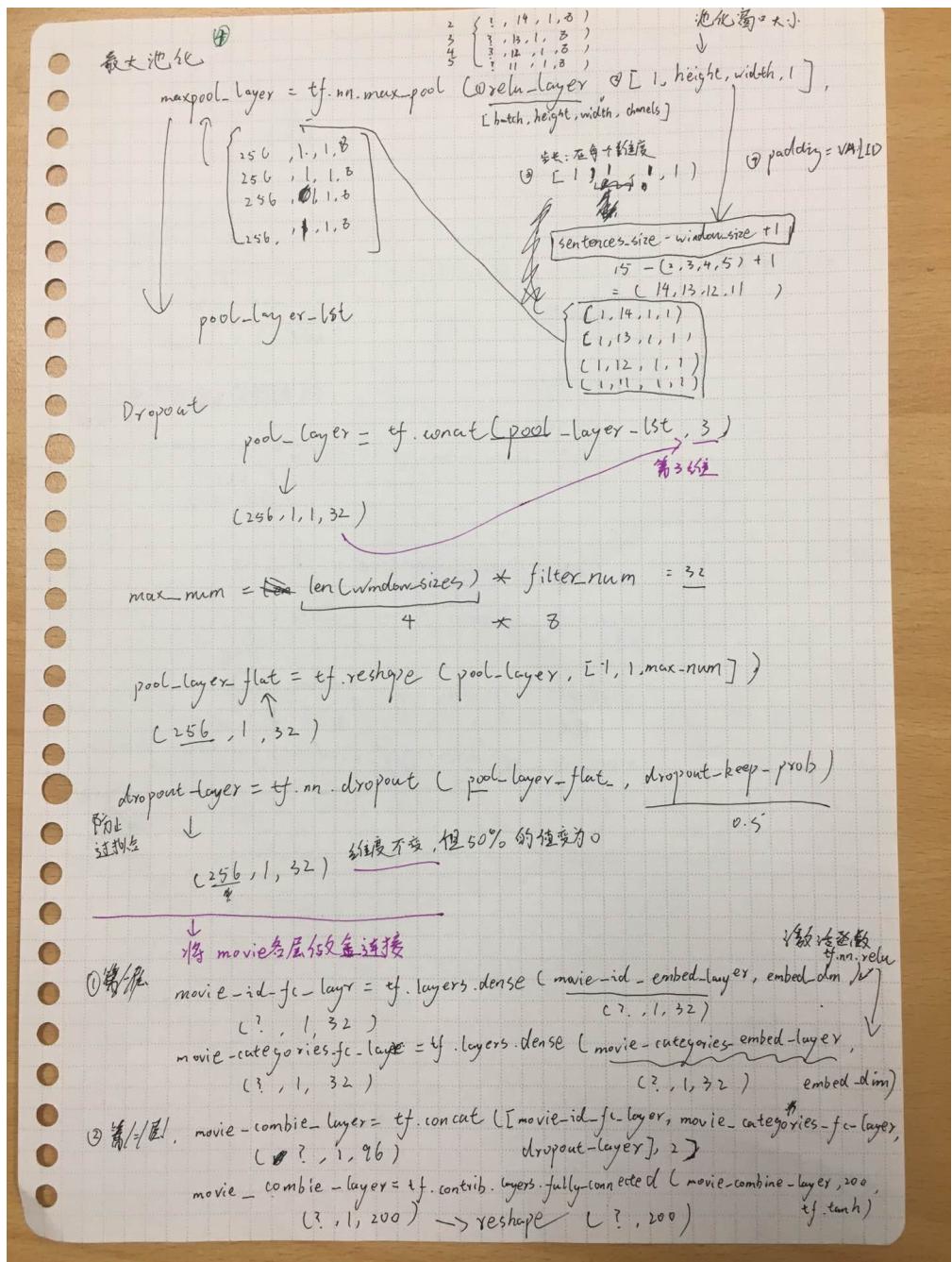
网络中有很多对MovieLens数据集的推荐算法，我想在学习了相关算法之后，能把这些算法用到工业界或者传统业会比针对一个已存在几十年的数据集提高那百分之零点几的准确率或降低微小的误差更有意义，当然，要解决的问题也会更多，加油吧！

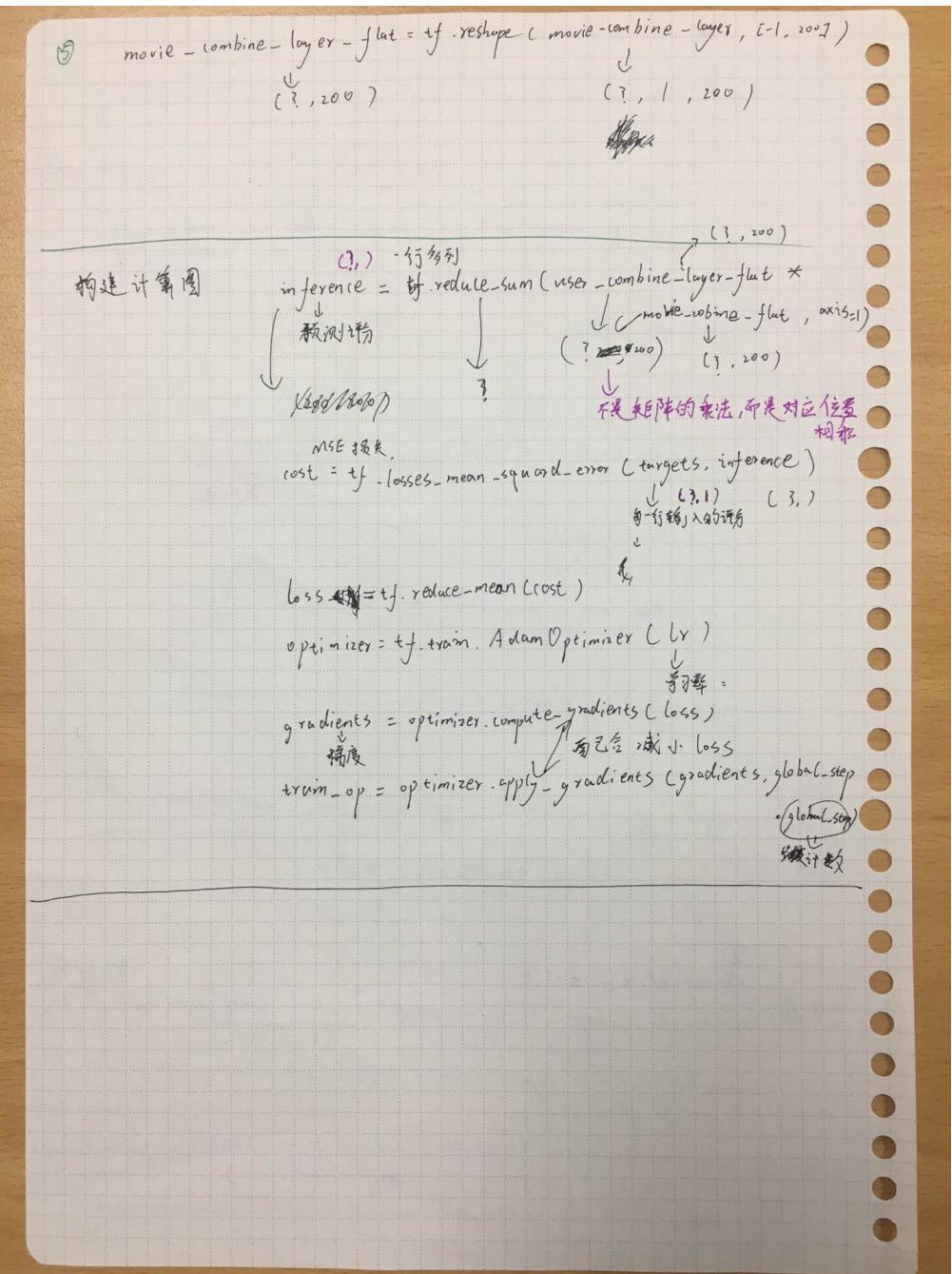
以下是我的推导手稿截图：

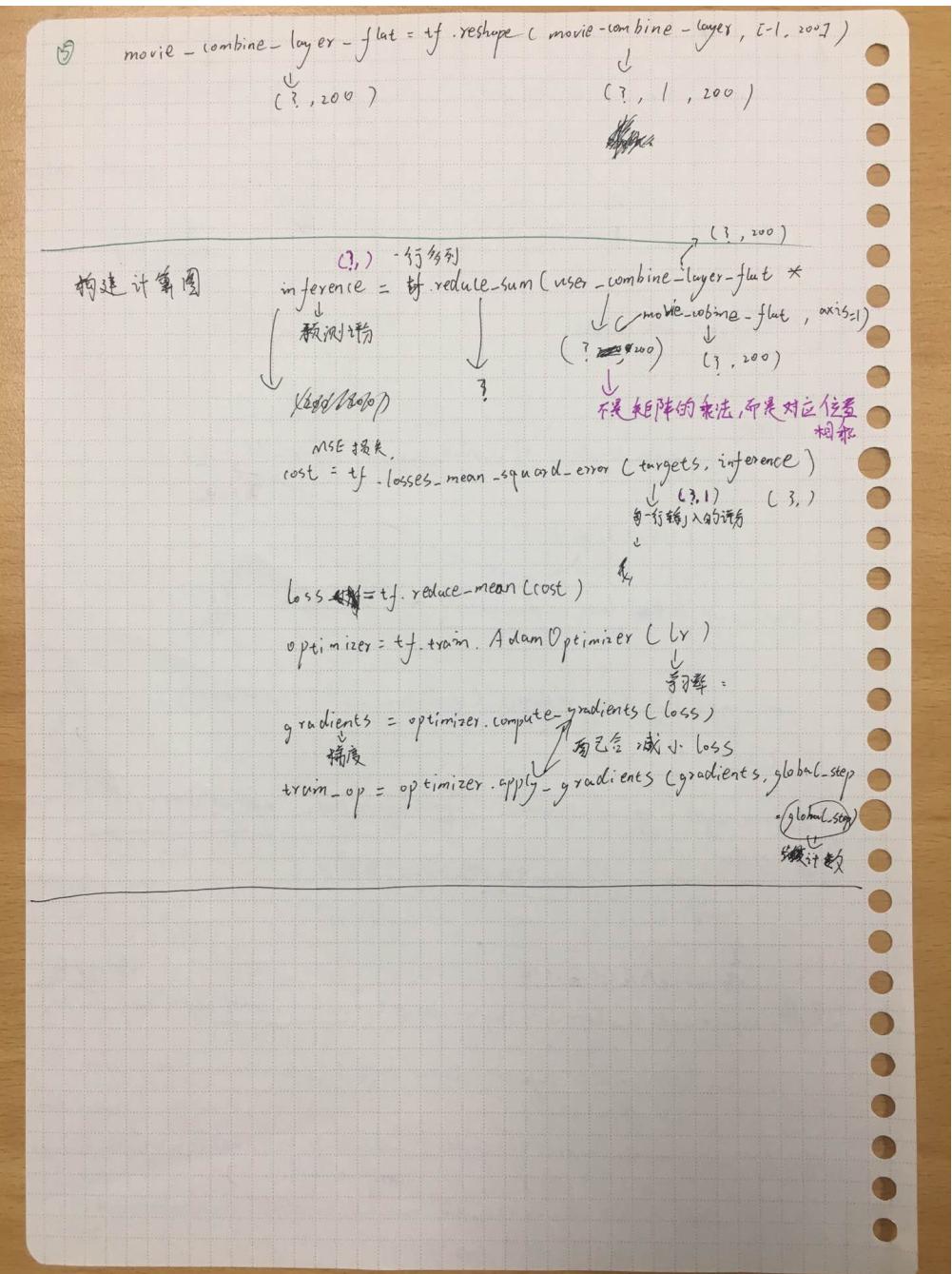












④ 训练：

rating-movie(id-val, movie-id-val) 网络正向传播

→ 简述 inference 图示

两矩阵相乘

得到预测得分

→ 将训练好的电影特征矩阵保存到本地

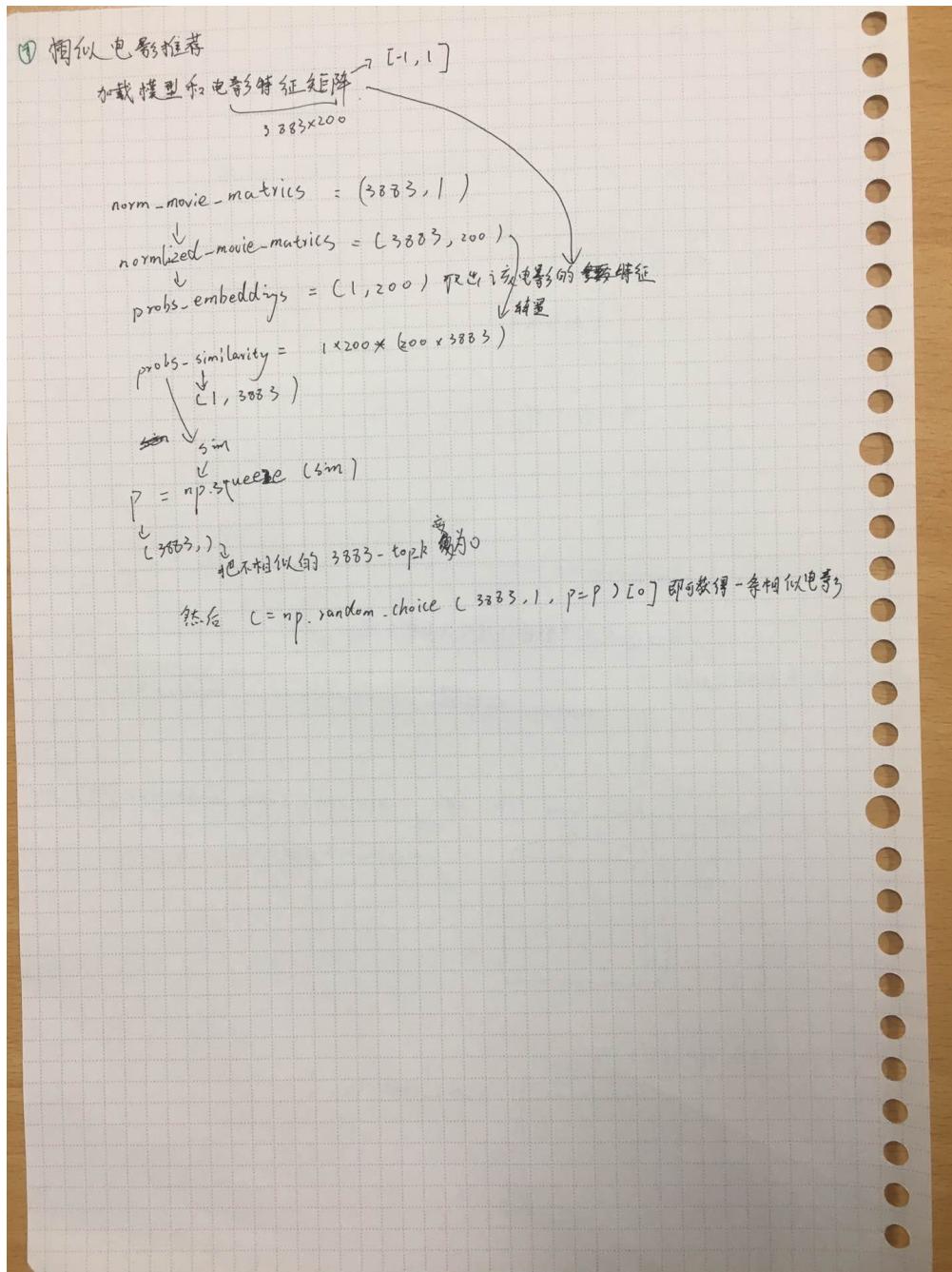
生成电影特征矩阵 { list-
list[0]: movie_id movie-categories movies-titles

movie-combine-layer-flat-val = sess.run([movie_combine_layer_flat], feed)
add (3, 200)

3883 { [] movies-matrices
{ { } }

同理 生成 用户特征矩阵 200

604 { [] users-matrices
{ { } }



五. 参考文献

[【1】Convolutional Neural Networks for Sentence Classification](#)

[【2】Understanding Convolutional Neural Networks for NLP](#)

六. 参考代码

原作者Github: [\[chengstone\]/movie recommender](#)