七月在线电商推荐系统项目特训营结业考试参考答案

By 魏天保

一、理论题（40分）

1、协同过滤算法是指基于用户行为数据设计的推荐算法，常见的有UserCF和ItemCF两种，请你分别用一句话概括这两种算法的核心思想。（10分）

答案：UserCF的核心思想是：给用户推荐和他兴趣相似的其他用户喜欢的物品。ItemCF的核心思想是：给用户推荐和其过去感兴趣的物品相似的物品。

2、 矩阵分解属于协同过滤算法吗？（5分）

A 属于

B 不属于

答案：A

3、请列举出基于内容的推荐的两个优点（6分）

答：（1）没有物品冷启动的问题（2）推荐结果直观，容易理解

4、请说出推荐系统中评分预测和TopN推荐的评价指标，各列举两种。（4分）

答案：评分预测：RMSE、MAE；TopN推荐：Precision、Recall。

5、 写出RMSE的计算公式，并说明公式中每个变量的含义。（5分）

答案：

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

m是测试集样本个数，y_i是第 i 个测试样本真实值，$\hat{y}_i$是第 i 个测试样本预测值

6、Deep AutoEncoder 的核心思想是什么。（10分）

答案：核心的思想是通过反向传播训练网络，最小化输入和输出的误差，从而完成对未知的电影的预估。

二、代码题（60分）

7、请按要求完成基于 tensorflow 的 Deep AutoEncoder 算法。（共30分）

In [1]:

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import tensorflow as tf
```

加载数据

In [2]:

```
df = pd.read_csv('ratings.dat', sep='\t', names=['user', 'item', 'rating', 'timestamp'], header
```

(1) 去掉评分数据 df 中的 “timestamp” 这一列，并输出 df 前五行（5分）

In [3]:

```
df = df.drop('timestamp', axis=1)
df.head()
```

Out[3]:

	user	item	rating
0	1	1193	5
1	1	661	3
2	1	914	3
3	1	3408	4
4	1	2355	5

查看用户数和电影数

In [4]:

```
num_items = df.item.nunique()
num_users = df.user.nunique()
print("USERS: {} ITEMS: {}".format(num_users, num_items))
```

```
USERS: 6040 ITEMS: 3706
```

(2) 对 df 的 “rating” 列做 Normalization，展示 df 前 5 行（5分）

In [5]:

```
from sklearn import preprocessing
r = df['rating'].values.astype(float)
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(r.reshape(-1,1))
df_normalized = pd.DataFrame(x_scaled)
df['rating'] = df_normalized
df.head()
```

Out[5]:

	user	item	rating
0	1	1193	1.00
1	1	661	0.50
2	1	914	0.50
3	1	3408	0.75
4	1	2355	1.00

(3) 把 df 转成 user-item 矩阵，存储在变量 matrix 中 ， 用 0 填充矩阵的缺失值，从matrix中取出user5对item6的评分值（5分）

In [6]:

```
matrix = df.pivot(index='user', columns='item', values='rating')
matrix.fillna(0, inplace=True)
matrix.loc[5,6]
```

Out[6]:

0.25

获取用户和物品列表

In [7]:

```
users = matrix.index.tolist()
items = matrix.columns.tolist()
```

将 matrix 从 dataframe 转换成 numpy 数组

In [8]:

```
matrix = matrix.as_matrix()
```

网络超参数

In [9]:

```
num_input = num_items
num_hidden_1 = 10
num_hidden_2 = 5
```

隐层的变量初始化

In [10]:

```
weights = {
    'encoder_h1': tf.Variable(tf.random_normal([num_input, num_hidden_1], dtype=tf.float64)),
    'encoder_h2': tf.Variable(tf.random_normal([num_hidden_1, num_hidden_2], dtype=tf.float64))
    'decoder_h1': tf.Variable(tf.random_normal([num_hidden_2, num_hidden_1], dtype=tf.float64))
    'decoder_h2': tf.Variable(tf.random_normal([num_hidden_1, num_input], dtype=tf.float64)),
}

biases = {
    'encoder_b1': tf.Variable(tf.random_normal([num_hidden_1], dtype=tf.float64)),
    'encoder_b2': tf.Variable(tf.random_normal([num_hidden_2], dtype=tf.float64)),
    'decoder_b1': tf.Variable(tf.random_normal([num_hidden_1], dtype=tf.float64)),
    'decoder_b2': tf.Variable(tf.random_normal([num_input], dtype=tf.float64)),
}
```

(4) 构建 encoder 和 decoder （5分）

In [11]:

```
def encoder(x):
    layer_1 = tf.nn.relu(tf.add(tf.matmul(x, weights['encoder_h1']), biases['encoder_b1']))
    layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, weights['encoder_h2']), biases['encoder_b2']
    return layer_2

def decoder(x):
    layer_1 = tf.nn.relu(tf.add(tf.matmul(x, weights['decoder_h1']), biases['decoder_b1']))
    layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, weights['decoder_h2']), biases['decoder_b2']
    return layer_2
```

构建整个模型

In [12]:

```
X = tf.placeholder(tf.float64, [None, num_input])
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)
```

预测 y 值和真实 y 值

In [13]:

```
y_pred = decoder_op
y_true = X
```

定义损失函数和优化器

In [14]:

```
loss = tf.losses.mean_squared_error(y_true, y_pred)
optimizer = tf.train.RMSPropOptimizer(0.03).minimize(loss)
```

定义评估准则

In [15]:

```
eval_x = tf.placeholder(tf.int32, )
eval_y = tf.placeholder(tf.int32, )
pre, pre_op = tf.metrics.precision(labels=eval_x, predictions=eval_y)
```

变量初始化

In [16]:

```
init = tf.global_variables_initializer()
local_init = tf.local_variables_initializer()
```

在 session 中 run

In [17]:

```
predictions = pd.DataFrame()
with tf.Session() as session:
    epochs = 100
    batch_size = 250

    session.run(init)
    session.run(local_init)

    num_batches = int(matrix.shape[0] / batch_size)
    matrix = np.array_split(matrix, num_batches)

    for i in range(epochs):

        avg_cost = 0

        for batch in matrix:
            _, l = session.run([optimizer, loss], feed_dict={X: batch})
            avg_cost += l

        avg_cost /= num_batches

        print("Epoch: {} Loss: {}".format(i + 1, avg_cost))

    print("Predictions...")

    matrix = np.concatenate(matrix, axis=0)

    preds = session.run(decoder_op, feed_dict={X: matrix})

    predictions = predictions.append(pd.DataFrame(preds))

    predictions = predictions.stack().reset_index(name='rating')
    predictions.columns = ['user', 'item', 'rating']
    predictions['user'] = predictions['user'].map(lambda value: users[value])
    predictions['item'] = predictions['item'].map(lambda value: items[value])
```

```
Epoch: 1 Loss: 31.396583735594284
Epoch: 2 Loss: 1.4016507441798847
Epoch: 3 Loss: 0.6289405835171541
Epoch: 4 Loss: 0.47894836962223053
Epoch: 5 Loss: 0.41464972496032715
Epoch: 6 Loss: 0.2824529707431793
Epoch: 7 Loss: 0.12261824092517297
Epoch: 8 Loss: 0.04141972423531115
Epoch: 9 Loss: 0.023202844196930528
Epoch: 10 Loss: 0.020845729508437216
Epoch: 11 Loss: 0.020702326049407322
Epoch: 12 Loss: 0.02070566701392333
Epoch: 13 Loss: 0.020706364419311285
Epoch: 14 Loss: 0.020706464264852304
Epoch: 15 Loss: 0.02070651645772159
Epoch: 16 Loss: 0.020706516418916483
Epoch: 17 Loss: 0.020706516263696056
Epoch: 18 Loss: 0.020706516263696056
Epoch: 19 Loss: 0.02070651634130627
Epoch: 20 Loss: 0.02070651634130627
Epoch: 21 Loss: 0.020706516263696056
Epoch: 22 Loss: 0.02070651634130627
```

```
Epoch: 23 Loss: 0.02070651634130627
Epoch: 24 Loss: 0.02070651634130627
Epoch: 25 Loss: 0.02070651634130627
Epoch: 26 Loss: 0.02070651634130627
Epoch: 27 Loss: 0.02070651634130627
Epoch: 28 Loss: 0.020706516418916483
Epoch: 29 Loss: 0.02070651634130627
Epoch: 30 Loss: 0.020706516263696056
Epoch: 31 Loss: 0.020706516263696056
Epoch: 32 Loss: 0.020706516418916483
Epoch: 33 Loss: 0.02070651634130627
Epoch: 34 Loss: 0.02070651634130627
Epoch: 35 Loss: 0.02070651634130627
Epoch: 36 Loss: 0.020706516263696056
Epoch: 37 Loss: 0.02070651634130627
Epoch: 38 Loss: 0.02070651634130627
Epoch: 39 Loss: 0.02070651634130627
Epoch: 40 Loss: 0.02070651634130627
Epoch: 41 Loss: 0.02070651634130627
Epoch: 42 Loss: 0.02070651634130627
Epoch: 43 Loss: 0.02070651634130627
Epoch: 44 Loss: 0.02070651634130627
Epoch: 45 Loss: 0.020706516263696056
Epoch: 46 Loss: 0.02070651634130627
Epoch: 47 Loss: 0.02070651634130627
Epoch: 48 Loss: 0.020706516263696056
Epoch: 49 Loss: 0.020706516263696056
Epoch: 50 Loss: 0.02070651634130627
Epoch: 51 Loss: 0.02070651634130627
Epoch: 52 Loss: 0.02070651634130627
Epoch: 53 Loss: 0.020706516263696056
Epoch: 54 Loss: 0.02070651634130627
Epoch: 55 Loss: 0.020706516263696056
Epoch: 56 Loss: 0.02070651634130627
Epoch: 57 Loss: 0.020706516263696056
Epoch: 58 Loss: 0.020706516263696056
Epoch: 59 Loss: 0.02070651634130627
Epoch: 60 Loss: 0.020706516263696056
Epoch: 61 Loss: 0.02070651634130627
Epoch: 62 Loss: 0.02070651634130627
Epoch: 63 Loss: 0.020706516263696056
Epoch: 64 Loss: 0.020706516263696056
Epoch: 65 Loss: 0.02070651634130627
Epoch: 66 Loss: 0.020706516263696056
Epoch: 67 Loss: 0.020706516263696056
Epoch: 68 Loss: 0.020706516418916483
Epoch: 69 Loss: 0.020706516263696056
Epoch: 70 Loss: 0.020706516263696056
Epoch: 71 Loss: 0.02070651634130627
Epoch: 72 Loss: 0.02070651634130627
Epoch: 73 Loss: 0.020706516418916483
Epoch: 74 Loss: 0.020706516418916483
Epoch: 75 Loss: 0.02070651634130627
Epoch: 76 Loss: 0.02070651634130627
Epoch: 77 Loss: 0.020706516263696056
Epoch: 78 Loss: 0.02070651634130627
Epoch: 79 Loss: 0.020706516263696056
Epoch: 80 Loss: 0.02070651634130627
Epoch: 81 Loss: 0.02070651634130627
Epoch: 82 Loss: 0.02070651634130627
Epoch: 83 Loss: 0.02070651634130627
```

```
Epoch: 84 Loss: 0.02070651634130627
Epoch: 85 Loss: 0.020706516418916483
Epoch: 86 Loss: 0.02070651634130627
Epoch: 87 Loss: 0.020706516263696056
Epoch: 88 Loss: 0.020706516263696056
Epoch: 89 Loss: 0.020706516418916483
Epoch: 90 Loss: 0.02070651634130627
Epoch: 91 Loss: 0.02070651634130627
Epoch: 92 Loss: 0.020706516263696056
Epoch: 93 Loss: 0.020706516263696056
Epoch: 94 Loss: 0.020706516263696056
Epoch: 95 Loss: 0.020706516418916483
Epoch: 96 Loss: 0.02070651634130627
Epoch: 97 Loss: 0.02070651634130627
Epoch: 98 Loss: 0.02070651634130627
Epoch: 99 Loss: 0.020706516263696056
Epoch: 100 Loss: 0.020706516263696056
Predictions...
```

(5) 为用户42计算top10的推荐结果（10分）

- top10 的筛选依据是评分预测最高的 top10
- 要求推荐的 top10 是该用户没有看过的电影
- 最后输出推荐的 top10 电影的 itemid

In [18]:

```
seen_movies = df[df.user==42].item.values
rec_for_42 = predictions[(predictions.user==42)&(~predictions.item.isin(seen_movies))]
top10 = rec_for_42.sort_values('rating',ascending=False).head(10).item.values
print(top10)
```

```
[2396  110    1  912   50 1247 1259  111 1225 2324]
```

8、请你实现基于Keras的协同深度学习算法。（共30分）

In [1]:

```
import numpy as np
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from keras.layers import Input, Embedding, Dot, Flatten, Dense, Dropout, Lambda, Add
from keras.layers.noise import GaussianNoise
from keras.initializers import RandomUniform, RandomNormal
from keras.models import Model
from keras.regularizers import l2
from keras import optimizers
from keras import backend as K
from keras.engine.topology import Layer
```

```
Using TensorFlow backend.
```

(1) 请你完成 movie_map 这个函数（5分）

- 功能：使用自然数对 movies.dat 中的 movieid 进行重编码
- 返回值：movieid 到编码 id 的字典
- 测试：计算编码字典，存入变量 d（后面有用），输出 d[520]

In [2]:

```
def movie_map(file='movies.dat'):
    movies = read_csv(file, sep='::', header=None,engine='python')
    value = movies[0].unique()
    index = range(movies[0].nunique())
    return dict(zip(value,index))

d = movie_map()
d[520]
```

Out[2]:

```
516
```

(2) 请你完成 read_ratings 这个函数（5分）

- 要求返回数据类型为 float32 的二维 numpy 数组：每一行表示用户评分记录 [user,item,rating]
- 原始数据中的 itemid 需要使用编码字典 d 进行重编码
- rating 值需要进行幅度缩放，缩放区间为[0,1]
- 测试：调用 read_ratings ,将返回值存入变量 ratings （后面有用），并输出 ratings[:3]

In [3]:

```
def read_ratings(file='ratings.dat'):
    rating_mat = list()
    with open(file) as fp:
        for line in fp:
            line = line.strip().split('\t')
            user, item, rating = line[0], d[int(line[1])], int(line[2])/5
            rating_mat.append( [user, item, rating] )
    return np.array(rating_mat).astype('float32')

ratings = read_ratings()
ratings[:3]
```

Out[3]:

```
array([[1.000e+00, 1.176e+03, 1.000e+00],
       [1.000e+00, 6.550e+02, 6.000e-01],
       [1.000e+00, 9.020e+02, 6.000e-01]], dtype=float32)
```

(3) 请你完成 train_test_split 这个函数（5分）

- 功能：将评分数据 ratings 划分为训练集和测试集两部分
- 输入：函数 read_ratings 的返回值 ratings
- 输出：训练集 train_mat，测试集 test_mat
- 要求：训练集占 80%，测试集占 20%
- 测试：调用该函数，返回值存入 train_mat, test_mat （后面有用），并展示训练集和测试集的样本数量

In [4]:

```
def train_test_split(ratings):

    data_num = len(ratings)
    train_mat = ratings[:int(data_num*0.8)]
    test_mat = ratings[int(data_num*0.8):]

    return train_mat, test_mat

train_mat, test_mat = train_test_split(ratings)
train_mat.shape[0], test_mat.shape[0]
```

Out[4]:

```
(800167, 200042)
```

(4) 请你完成 read_item 这个函数（5分）

- 功能：将 movies.dat 的三列特征进行重编码
- 要求：
  - 第 1 列特征使用编码字典 d 进行编码
  - 第 2、3 列特征使用 LabelEncoder() 编码
  - 返回值有两个，第一个是二维 numpy 数组 ， 也就是是一个矩阵：每一行对应原始数据 movies.dat 的一行记录；第二个是特征维度，也就是矩阵的列数
  - 调用该函数，返回值存入 item_mat,item_feat_dim （后面有用），打印 item_mat[:3] 和 item_feat_dim

In [5]:

```
def read_item(file='movies.dat'):

    item = read_csv(file, sep='::', header=None,engine='python')
    item[0] = item[0].apply(lambda x:d[int(x)])
    item[1] = LabelEncoder().fit_transform(item[1])
    item[2] = LabelEncoder().fit_transform(item[2])
    item_mat = item.as_matrix()
    item_feat_dim = item_mat.shape[1]

    return item_mat,item_feat_dim

item_mat,item_feat_dim = read_item()
print(item_mat[:3])
print(item_feat_dim)
```

```
[[   0 3574  145]
 [   1 1858  115]
 [   2 1483  207]]
3
```

CollaborativeDeepLearning 实现

In [6]:

```
class CollaborativeDeepLearning:

    def __init__(self, item_mat, hidden_layers):
        '''
        hidden_layers = a list of three integer indicating the embedding dimension of autoencod
        item_mat = item feature matrix with shape (# of item, # of item features)
        '''
        assert(len(hidden_layers)==3)
        self.item_mat = item_mat
        self.hidden_layers = hidden_layers
        self.item_dim = hidden_layers[0]
        self.embedding_dim = hidden_layers[-1]

    def pretrain(self, lamda_w=0.1, encoder_noise=0.1, dropout_rate=0.1, activation='sigmoid',
        '''
        layer-wise pretraining on item features (item_mat)
        '''
        self.trained_encoders = []
        self.trained_decoders = []
        X_train = self.item_mat

        for input_dim, hidden_dim in zip(self.hidden_layers[:-1], self.hidden_layers[1:]):

            pretrain_input = Input(shape=(input_dim,))
            encoded = GaussianNoise(stddev=encoder_noise)(pretrain_input)
            encoded = Dropout(dropout_rate)(encoded)
            encoder = Dense(hidden_dim, activation=activation, kernel_regularizer=l2(lamda_w),
            decoder = Dense(input_dim, activation=activation, kernel_regularizer=l2(lamda_w), l

            # autoencoder
            ae = Model(inputs=pretrain_input, outputs=decoder)

            # encoder
            ae_encoder = Model(inputs=pretrain_input, outputs=encoder)

            # decoder
            encoded_input = Input(shape=(hidden_dim,))

            decoder_layer = ae.layers[-1] # the last layer
            ae_decoder = Model(encoded_input, decoder_layer(encoded_input))

            ae.compile(loss='mse', optimizer='rmsprop')
            ae.fit(X_train, X_train, batch_size=batch_size, epochs=epochs, verbose=2)

            self.trained_encoders.append(ae_encoder)
            self.trained_decoders.append(ae_decoder)
            X_train = ae_encoder.predict(X_train)

    def fineture(self, train_mat, test_mat, lamda_u=0.1, lamda_v=0.1, lamda_n=0.1, lr=0.001, b
        '''
        Fine-tuning with rating prediction
        '''
        num_user = int( max(train_mat[:,0].max(), test_mat[:,0].max()) + 1 )
        num_item = int( max(train_mat[:,1].max(), test_mat[:,1].max()) + 1 )

        # item autoencoder
        itemfeat_InputLayer = Input(shape=(self.item_dim,), name='item_feat_input')
        encoded = self.trained_encoders[0](itemfeat_InputLayer)
        encoded = self.trained_encoders[1](encoded)
```

```
        decoded = self.trained_decoders[1](encoded)
        decoded = self.trained_decoders[0](decoded)

        # user embedding
        user_InputLayer = Input(shape=(1,), dtype='int32', name='user_input')
        user_EmbeddingLayer = Embedding(input_dim=num_user, output_dim=self.embedding_dim,
                                        input_length=1, name='user_embedding',
                                        embeddings_regularizer=l2(lamda_u),
                                        embeddings_initializer=RandomNormal(mean=0, stddev=1))
        user_EmbeddingLayer = Flatten(name='user_flatten')(user_EmbeddingLayer)

        # item embedding
        item_InputLayer = Input(shape=(1,), dtype='int32', name='item_input')
        item_OffsetVector = Embedding(input_dim=num_item, output_dim=self.embedding_dim,
                                      input_length=1, name='item_offset_vector',
                                      embeddings_regularizer=l2(lamda_v),
                                      embeddings_initializer=RandomNormal(mean=0, stddev=1))(i
        item_OffsetVector = Flatten(name='item_flatten')(item_OffsetVector)
        item_EmbeddingLayer = Add()([encoded, item_OffsetVector])

        # rating prediction
        dotLayer = Dot(axes = -1, name='dot_layer')([user_EmbeddingLayer, item_EmbeddingLayer]

        self.cdl_model = Model(inputs=[user_InputLayer, item_InputLayer, itemfeat_InputLayer],
        self.cdl_model.compile(optimizer='rmsprop', loss=['mse', 'mse'], loss_weights=[1, lamda

        train_user, train_item, train_item_feat, train_label = self.matrix2input(train_mat)
        test_user, test_item, test_item_feat, test_label = self.matrix2input(test_mat)

        model_history = self.cdl_model.fit([train_user, train_item, train_item_feat],
                                           [train_label, train_item_feat],
                                           epochs=epochs, batch_size=batch_size,
                                           validation_data=([test_user, test_item, test_item_fe
        return model_history

    def matrix2input(self, rating_mat):
        train_user = rating_mat[:, 0].reshape(-1, 1).astype(int)
        train_item = rating_mat[:, 1].reshape(-1, 1).astype(int)
        train_label = rating_mat[:, 2].reshape(-1, 1)
        train_item_feat = [self.item_mat[train_item[x]][0] for x in range(train_item.shape[0])
        return train_user, train_item, np.array(train_item_feat), train_label

    def build(self):
        # rating prediction
        prediction_layer = Dot(axes = -1, name='prediction_layer')([user_EmbeddingLayer, encod
        self.model = Model(inputs=[user_InputLayer, itemfeat_InputLayer], outputs=[prediction_

    def getRMSE(self, test_mat):
        test_user, test_item, test_item_feat, test_label = self.matrix2input(test_mat)
        pred_out = self.cdl_model.predict([test_user, test_item, test_item_feat])
        return np.sqrt(np.mean(np.square(test_label.flatten() - pred_out[0].flatten())))
```

(5) 成功运行下面的代码，输出RMSE（10分）

In [7]:

```
model = CollaborativeDeepLearning(item_mat, [item_feat_dim, 16, 8])
model.pretrain(lamda_w=0.001, encoder_noise=0.3, epochs=10)
model_history = model.fineture(train_mat, test_mat, lamda_u=0.01, lamda_v=0.1, lamda_n=0.1, lr
model.getRMSE(test_mat)
```

```
Epoch 1/10
0s - loss: 3362782.6749
Epoch 2/10
0s - loss: 3362457.1563
Epoch 3/10
0s - loss: 3362184.4979
Epoch 4/10
0s - loss: 3361981.1230
Epoch 5/10
0s - loss: 3361836.0971
Epoch 6/10
0s - loss: 3361736.3306
Epoch 7/10
0s - loss: 3361669.6875
Epoch 8/10
0s - loss: 3361630.6741
Epoch 9/10
0s - loss: 3361608.1432
Epoch 10/10
0s - loss: 3361595.3847
Epoch 1/10
0s - loss: 0.2609
Epoch 2/10
0s - loss: 0.2068
Epoch 3/10
0s - loss: 0.1644
Epoch 4/10
0s - loss: 0.1371
Epoch 5/10
0s - loss: 0.1213
Epoch 6/10
0s - loss: 0.1124
Epoch 7/10
0s - loss: 0.1070
Epoch 8/10
0s - loss: 0.1039
Epoch 9/10
0s - loss: 0.1019
Epoch 10/10
0s - loss: 0.1005
Train on 800167 samples, validate on 200042 samples
Epoch 1/4
800167/800167 [==============================] - 47s - loss: 320012.3428 - dot_layer
_loss: 0.7405 - model_3_loss: 3198583.5023 - val_loss: 311579.5984 - val_dot_layer_l
oss: 0.5741 - val_model_3_loss: 3115789.5686
Epoch 2/4
800167/800167 [==============================] - 44s - loss: 319858.8802 - dot_layer
_loss: 0.4799 - model_3_loss: 3198583.4552 - val_loss: 311579.5988 - val_dot_layer_l
oss: 0.5745 - val_model_3_loss: 3115789.5686
Epoch 3/4
800167/800167 [==============================] - 48s - loss: 319858.8803 - dot_layer
_loss: 0.4800 - model_3_loss: 3198583.4564 - val_loss: 311579.5697 - val_dot_layer_l
oss: 0.5741 - val_model_3_loss: 3115789.5686
Epoch 4/4
```

```
800167/800167 [==============================] - 52s - loss: 319858.8798 - dot_layer
_loss: 0.4800 - model_3_loss: 3198583.4517 - val_loss: 311579.5988 - val_dot_layer_l
oss: 0.5745 - val_model_3_loss: 3115789.5686
```

Out[7]:

```
0.75795275
```