# 计算广告CTR预估系列(二)–DeepFM实践

# 0. 变量说明

为了方便后面的阅读，我们先说明一下各个名称的含义：

1. field\_size: 输入X在进行one-hot之前的特征维度

2. feature\_size: 输入X在one-hot之后的特征维度， 又记作 n

3. embedding\_size: one-hot后的输入，进行嵌入后的维度，又记作 k

4. 代码中tf中维度None表示任意维度，我们用来表示输入样本的数量这一维度。

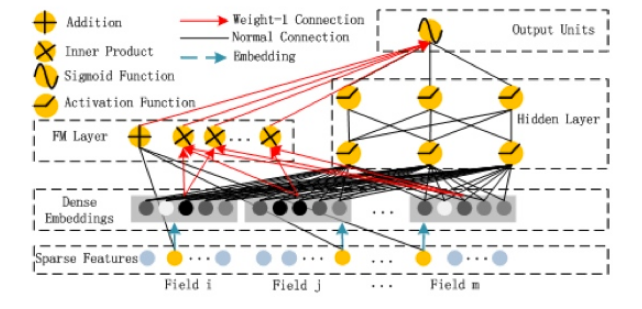
# 1. 架构图与公式

## 1.1 架构图

让我们来先回顾一下架构图和公式：

架构图包含两部分：FM Component和Deep Component。

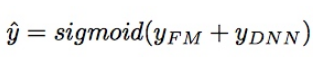
其中FM部分用于对1维特征和2维组合特征进行建模；Deep部分用于对高维组合特征进行建模。



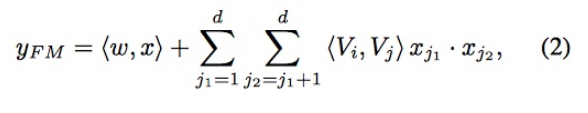
## 1.2 公式

### 1.2.1 公式参考

DeepFM最后输出公式为：



其中FM部分贡献为：



需要注意的点：

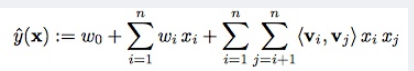
1. 这里省去了原始FM中的常数项，为了方便。

2. 这里的x可以认为是一个样本，d是one-hot之后的总维度，也就是feature\_size. 在原始FM论文中对应n参数。

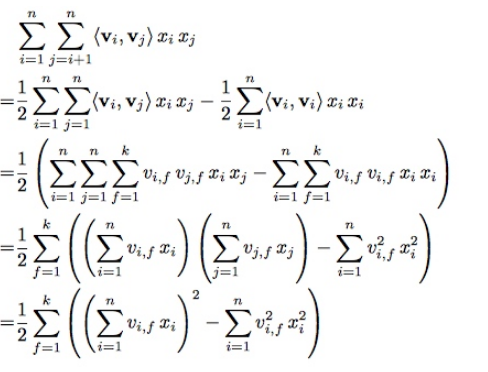
Deep部分贡献为：

这个很好理解，就是神经网络的输出而已，就不再给公式了。

**FM论文中原始公式，方便对比参考：**



二阶项的化简结果：



### 1.2.2 FM Component维度问题

先说结论：

* FM-1维：field\_size
* FM-2维组合特征： embedding\_size
* 这个和上面的公式有点不一样，但是实际代码实现是这样实现的。

这里比较重要：因为关系到代码里面怎么写，下面分别解释：

**先看FM-1次项：**

首先，我们看到一次项是求和了的。但是在实际代码中，我们并没有对一次项求和。而是输出一个维度为k的向量，这里的k就是embedding\_size。

为什么要这么做那？我个人的感觉是：提高最后模型的学习效果。

最后神经网络的输出其实可以看做是logistic regression。它的输入由三部分组成：FM-1维，FM-二维，Deep部分。 自然而然，我们不希望FM-1维只是一个标量，一个数吧。这显然不利于LR模型的学习，那么如果用原始的维度那：公式里是Wi \* Xi，Xi的维度是n，n是one-hot之后的维度，这个维度太大了以至于我们才想出了各种办法来解决这个问题，用n显然不行。所以，就用field\_size。从逻辑上来说，也是对one-hot之前每个特征维度的一种建模。

**再看FM-2维组合特征部分：**

在上节中，我们看到了FM二阶项的化简结果。最外层是在Embeeding\_size上的求和。不做这个求和，而是得到一个embedding\_size维度的向量，就是送到最后输出单元的FM的二阶部分。

原因我想跟上面FM-1维的是一样的，求和之后就变成了一个标量，显然不利于后面的学习。

这两部分理解了之后，让我们来看下代码吧！

# 2. 核心代码拆解

## 2.1 输入

feat\_index = tf.placeholder(dtype=tf.int32, shape=[None, config.field\_size], name='feat\_index')

# [None, field\_size]

feat\_value = tf.placeholder(dtype=tf.float32, shape=[None, None], name='feat\_value')

# [None, field\_size]

label = tf.placeholder(dtype=tf.float16, shape=[None,1], name='label')

注意下各个输入变量的维度大小就可以了，没什么特别需要说明的。None在tf里面表示任意维度，此处表示样本数量维度。

## 2.2 Embedding

# Sparse Features -> Dense Embedding

embeddings\_origin = tf.nn.embedding\_lookup(weights['feature\_embedding'], ids=feat\_index) # [None, field\_size, embedding\_size]

重点来了，这里是完成Sparse Features到Dense Embedding的转换。

## 2.3 FM Component - 1维特征

1维特征本来是求和得到一个标量的，为了提高学习效果，我们改为不求和，得到一个field\_size维度的向量。相当于是进行了一次k=1的一次embedding。

y\_first\_order = tf.nn.embedding\_lookup(weights['feature\_bias'], ids=feat\_index)

# [None, field\_size, 1]

得到W之后，和输入X相乘，并缩减维度，得到最终FM-1维的输出：

w\_mul\_x = tf.multiply(y\_first\_order, feat\_value\_reshape) # [None, field\_size, 1] Wi \* Xi

y\_first\_order = tf.reduce\_sum(input\_tensor=w\_mul\_x, axis=2) # [None, field\_size]

## 2.4 FM Component - 2维组合特征

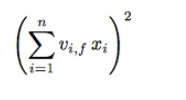
在第一节中，我们给出了FM-2维组合特征的化简公式。发现计算的两部分都需要计算 vi \* xi。所以我们先把这一部分计算出来：

feat\_value\_reshape = tf.reshape(tensor=feat\_value, shape=[-1, config.field\_size, 1]) # -1 \* field\_size \* 1

embeddings = tf.multiply(embeddings\_origin, feat\_value\_reshape)

# [None, field\_size, embedding\_size] multiply不是矩阵相乘，而是矩阵对应位置相乘。这里应用了broadcast机制。

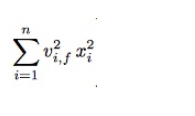
然后分别计算这两部分：



// sum\_square part 先sum，再square

summed\_features\_emb = tf.reduce\_sum(input\_tensor=embeddings, axis=1) # [None, embedding\_size]

summed\_features\_emb\_square = tf.square(summed\_features\_emb)



// square\_sum part

squared\_features\_emb = tf.square(embeddings)

squared\_features\_emb\_summed = tf.reduce\_sum(input\_tensor=squared\_features\_emb, axis=1) # [None, embedding\_size]

最后得到最终FM-2维组合特征输出结果，维度为embedding\_size:

// second order

y\_second\_order = 0.5 \* tf.subtract(summed\_features\_emb\_square, squared\_features\_emb\_summed)

## 2.5 Deep Component

网络部分比较简单，只要一层一层的前向传递就可以了。只有一个问题需要说明：

网络部分，第一个隐藏层的输入是什么？

我的感觉应该是原始输入嵌入后的结果：

// Deep Component

y\_deep = tf.reshape(embeddings\_origin, shape=[-1, config.field\_size \* config.embedding\_size]) # [None, field\_size \* embedding\_size]

for i in range(0, len(deep\_layers)):

y\_deep = tf.add(tf.matmul(y\_deep, weights['layer\_%d' % i]), weights['bias\_%d' % i])

y\_deep = config.deep\_layers\_activation(y\_deep)

## 2.6 输出

把前面FM Component和Deep Component的两部分结合起来，就得到了最后输出单元的输入，经过sigmoid函数激活就可以得到最终结果了。

// output

concat\_input = tf.concat([y\_first\_order, y\_second\_order, y\_deep], axis=1)

out = tf.add(tf.matmul(concat\_input, weights['concat\_projection']), weights['concat\_bias'])

out = tf.nn.sigmoid(out)

# 3. 完整代码：

这份代码最主要的目的是学习，直接应用于工程的话还需要做一些优化。比如batch normalization、stack train、batch train以及代码重构等。

本代码可以帮助你快速实验，实现DeepFM，掌握其原理。

另外，可直接运行。github地址–欢迎follow/star/contribute

另外，附上一份整理的 DeepFM架构图-实现篇 的图。主要是帮助大家理解各个参数的维度、权重weights的维度、需要学习的维度、FM Deep两部分输出的维度。 要想实现DeepFM，只要把每一部分的维度搞清楚，网络的架构搞清楚就问题不大了。

<https://github.com/gutouyu/ML_CIA/blob/master/DeepFM/DeepFM.py>

|  |
| --- |
|  |
|  | import gc  import numpy as np |
|  | import pandas as pd |
|  | import tensorflow as tf |
|  |  |
|  | ################################## |
|  | # 0. Functions |
|  | ################################## |
|  | class Config(object): |
|  | """ |
|  | 用来存储一些配置信息 |
|  | """ |
|  | def \_\_init\_\_(self): |
|  | self.feature\_dict = None |
|  | self.feature\_size = None |
|  | self.field\_size = None |
|  | self.embedding\_size = 8 |
|  |  |
|  | self.epochs = 20 |
|  | self.deep\_layers\_activation = tf.nn.relu |
|  |  |
|  | self.loss = "logloss" |
|  | self.l2\_reg = 0.1 |
|  | self.learning\_rate = 0.1 |
|  |  |
|  | def FeatureDictionary(dfTrain=None, dfTest=None, numeric\_cols=None, ignore\_cols=None): |
|  | """ |
|  | 目的是给每一个特征维度都进行编号。 |
|  | 1. 对于离散特征，one-hot之后每一列都是一个新的特征维度。所以，原来的一维度对应的是很多维度，编号也是不同的。 |
|  | 2. 对于连续特征，原来的一维特征依旧是一维特征。 |
|  |  |
|  | 返回一个feat\_dict，用于根据 原特征名称和特征取值 快速查询出 对应的特征编号。 |
|  |  |
|  | :param dfTrain: 原始训练集 |
|  | :param dfTest: 原始测试集 |
|  | :param numeric\_cols: 所有数值型特征 |
|  | :param ignore\_cols: 所有忽略的特征. 除了数值型和忽略的，剩下的全部认为是离散型 |
|  | :return: feat\_dict, feat\_size |
|  | 1. feat\_size: one-hot之后总的特征维度。 |
|  | 2. feat\_dict是一个{}， key是特征string的col\_name, value可能是编号（int），可能也是一个字典。 |
|  | 如果原特征是连续特征： value就是int，表示对应的特征编号； |
|  | 如果原特征是离散特征：value就是dict，里面是根据离散特征的 实际取值 查询 该维度的特征编号。 因为离散特征one-hot之后，一个取值就是一个维度， |
|  | 而一个维度就对应一个编号。 |
|  | """ |
|  | assert not (dfTrain is None), "train dataset is not set" |
|  | assert not (dfTest is None), "test dataset is not set" |
|  |  |
|  | # 编号肯定是要train test一起编号的 |
|  | df = pd.concat([dfTrain, dfTest], axis=0) |
|  |  |
|  | # 返回值 |
|  | feat\_dict = {} |
|  |  |
|  | # 目前为止的下一个编号 |
|  | total\_cnt = 0 |
|  |  |
|  | for col in df.columns: |
|  | if col in ignore\_cols: # 忽略的特征不参与编号 |
|  | continue |
|  |  |
|  | # 连续特征只有一个编号 |
|  | if col in numeric\_cols: |
|  | feat\_dict[col] = total\_cnt |
|  | total\_cnt += 1 |
|  | continue |
|  |  |
|  | # 离散特征，有多少个取值就有多少个编号 |
|  | unique\_vals = df[col].unique() |
|  | unique\_cnt = df[col].nunique() |
|  | feat\_dict[col] = dict(zip(unique\_vals, range(total\_cnt, total\_cnt + unique\_cnt))) |
|  | total\_cnt += unique\_cnt |
|  |  |
|  | feat\_size = total\_cnt |
|  | return feat\_dict, feat\_size |
|  |  |
|  | def parse(feat\_dict=None, df=None, has\_label=False): |
|  | """ |
|  | 构造FeatureDict，用于后面Embedding |
|  | :param feat\_dict: FeatureDictionary生成的。用于根据col和value查询出特征编号的字典 |
|  | :param df: 数据输入。可以是train也可以是test,不用拼接 |
|  | :param has\_label: 数据中是否包含label |
|  | :return: Xi, Xv, y |
|  | """ |
|  | assert not (df is None), "df is not set" |
|  |  |
|  | dfi = df.copy() |
|  |  |
|  | if has\_label: |
|  | y = df['target'].values.tolist() |
|  | dfi.drop(['id','target'],axis=1, inplace=True) |
|  | else: |
|  | ids = dfi['id'].values.tolist() # 预测样本的ids |
|  | dfi.drop(['id'],axis=1, inplace=True) |
|  |  |
|  | # dfi是Feature index,大小和dfTrain相同，但是里面的值都是特征对应的编号。 |
|  | # dfv是Feature value, 可以是binary(0或1), 也可以是实值float，比如3.14 |
|  | dfv = dfi.copy() |
|  |  |
|  | for col in dfi.columns: |
|  | if col in IGNORE\_FEATURES: # 用到的全局变量： IGNORE\_FEATURES, NUMERIC\_FEATURES |
|  | dfi.drop([col], axis=1, inplace=True) |
|  | dfv.drop([col], axis=1, inplace=True) |
|  | continue |
|  |  |
|  | if col in NUMERIC\_FEATURES: # 连续特征1个维度，对应1个编号，这个编号是一个定值 |
|  | dfi[col] = feat\_dict[col] |
|  | else: |
|  | # 离散特征。不同取值对应不同的特征维度，编号也是不同的。 |
|  | dfi[col] = dfi[col].map(feat\_dict[col]) |
|  | dfv[col] = 1.0 |
|  |  |
|  | # 取出里面的值 |
|  | Xi = dfi.values.tolist() |
|  | Xv = dfv.values.tolist() |
|  |  |
|  | del dfi, dfv |
|  | gc.collect() |
|  |  |
|  | if has\_label: |
|  | return Xi, Xv, y |
|  | else: |
|  | return Xi, Xv, ids |
|  |  |
|  | ################################## |
|  | # 1. 配置信息 |
|  | ################################## |
|  |  |
|  | train\_file = "./data/train.csv" |
|  | test\_file = "./data/test.csv" |
|  |  |
|  | IGNORE\_FEATURES = [ |
|  | 'id', 'target' |
|  | ] |
|  | CATEGORITAL\_FEATURES = [ |
|  | 'feat\_cat\_1', 'feat\_cat\_2' |
|  | ] |
|  | NUMERIC\_FEATURES = [ |
|  | 'feat\_num\_1', 'feat\_num\_2' |
|  | ] |
|  |  |
|  | config = Config() |
|  |  |
|  | ################################## |
|  | # 2. 读取文件 |
|  | ################################## |
|  | dfTrain = pd.read\_csv(train\_file) |
|  | dfTest = pd.read\_csv(test\_file) |
|  |  |
|  |  |
|  | ################################## |
|  | # 3. 准备数据 |
|  | ################################## |
|  |  |
|  | # FeatureDict |
|  | config.feature\_dict, config.feature\_size = FeatureDictionary(dfTrain=dfTrain, dfTest=dfTest, numeric\_cols=NUMERIC\_FEATURES, ignore\_cols=IGNORE\_FEATURES) |
|  |  |
|  | # Xi, Xv |
|  | Xi\_train, Xv\_train, y = parse(feat\_dict=config.feature\_dict, df=dfTrain, has\_label=True) |
|  | Xi\_test, Xv\_test, ids = parse(feat\_dict=config.feature\_dict, df=dfTest, has\_label=False) |
|  | config.field\_size = len(Xi\_train[0]) |
|  |  |
|  |  |
|  |  |
|  | ################################## |
|  | # 4. 建立模型 |
|  | ################################## |
|  |  |
|  | # 模型参数 |
|  | deep\_layers = [32,32] |
|  | config.embedding\_size = 8 |
|  | config.deep\_layers\_activation = tf.nn.relu |
|  |  |
|  | # BUILD THE WHOLE MODEL |
|  | tf.set\_random\_seed(2018) |
|  |  |
|  |  |
|  | # init\_weight |
|  | weights = dict() |
|  | # Sparse Features 到 Dense Embedding的全连接权重。[其实是Embedding] |
|  | weights['feature\_embedding'] = tf.Variable(initial\_value=tf.random\_normal(shape=[config.feature\_size, config.embedding\_size],mean=0,stddev=0.1), |
|  | name='feature\_embedding', |
|  | dtype=tf.float32) |
|  | # Sparse Featues 到 FM Layer中Addition Unit的全连接。 [其实是Embedding,嵌入后维度为1] |
|  | weights['feature\_bias'] = tf.Variable(initial\_value=tf.random\_uniform(shape=[config.feature\_size, 1],minval=0.0,maxval=1.0), |
|  | name='feature\_bias', |
|  | dtype=tf.float32) |
|  | # Hidden Layer |
|  | num\_layer = len(deep\_layers) |
|  | input\_size = config.field\_size \* config.embedding\_size |
|  | glorot = np.sqrt(2.0 / (input\_size + deep\_layers[0])) # glorot\_normal: stddev = sqrt(2/(fan\_in + fan\_out)) |
|  | weights['layer\_0'] = tf.Variable(initial\_value=tf.random\_normal(shape=[input\_size, deep\_layers[0]],mean=0,stddev=glorot), |
|  | dtype=tf.float32) |
|  | weights['bias\_0'] = tf.Variable(initial\_value=tf.random\_normal(shape=[1, deep\_layers[0]],mean=0,stddev=glorot), |
|  | dtype=tf.float32) |
|  | for i in range(1, num\_layer): |
|  | glorot = np.sqrt(2.0 / (deep\_layers[i - 1] + deep\_layers[i])) |
|  | # deep\_layer[i-1] \* deep\_layer[i] |
|  | weights['layer\_%d' % i] = tf.Variable(initial\_value=tf.random\_normal(shape=[deep\_layers[i - 1], deep\_layers[i]],mean=0,stddev=glorot), |
|  | dtype=tf.float32) |
|  | # 1 \* deep\_layer[i] |
|  | weights['bias\_%d' % i] = tf.Variable(initial\_value=tf.random\_normal(shape=[1, deep\_layers[i]],mean=0,stddev=glorot), |
|  | dtype=tf.float32) |
|  | # Output Layer |
|  | deep\_size = deep\_layers[-1] |
|  | fm\_size = config.field\_size + config.embedding\_size |
|  | input\_size = fm\_size + deep\_size |
|  | glorot = np.sqrt(2.0 / (input\_size + 1)) |
|  | weights['concat\_projection'] = tf.Variable(initial\_value=tf.random\_normal(shape=[input\_size,1],mean=0,stddev=glorot), |
|  | dtype=tf.float32) |
|  | weights['concat\_bias'] = tf.Variable(tf.constant(value=0.01), dtype=tf.float32) |
|  |  |
|  |  |
|  | # build\_network |
|  | feat\_index = tf.placeholder(dtype=tf.int32, shape=[None, config.field\_size], name='feat\_index') # [None, field\_size] |
|  | feat\_value = tf.placeholder(dtype=tf.float32, shape=[None, None], name='feat\_value') # [None, field\_size] |
|  | label = tf.placeholder(dtype=tf.float16, shape=[None,1], name='label') |
|  |  |
|  | # Sparse Features -> Dense Embedding |
|  | embeddings\_origin = tf.nn.embedding\_lookup(weights['feature\_embedding'], ids=feat\_index) # [None, field\_size, embedding\_size] |
|  |  |
|  | feat\_value\_reshape = tf.reshape(tensor=feat\_value, shape=[-1, config.field\_size, 1]) # -1 \* field\_size \* 1 |
|  |  |
|  | # --------- 一维特征 ----------- |
|  | y\_first\_order = tf.nn.embedding\_lookup(weights['feature\_bias'], ids=feat\_index) # [None, field\_size, 1] |
|  | w\_mul\_x = tf.multiply(y\_first\_order, feat\_value\_reshape) # [None, field\_size, 1] Wi \* Xi |
|  | y\_first\_order = tf.reduce\_sum(input\_tensor=w\_mul\_x, axis=2) # [None, field\_size] |
|  |  |
|  | # --------- 二维组合特征 ---------- |
|  | embeddings = tf.multiply(embeddings\_origin, feat\_value\_reshape) # [None, field\_size, embedding\_size] multiply不是矩阵相乘，而是矩阵对应位置相乘。这里应用了broadcast机制。 |
|  |  |
|  | # sum\_square part 先sum，再square |
|  | summed\_features\_emb = tf.reduce\_sum(input\_tensor=embeddings, axis=1) # [None, embedding\_size] |
|  | summed\_features\_emb\_square = tf.square(summed\_features\_emb) |
|  |  |
|  | # square\_sum part |
|  | squared\_features\_emb = tf.square(embeddings) |
|  | squared\_features\_emb\_summed = tf.reduce\_sum(input\_tensor=squared\_features\_emb, axis=1) # [None, embedding\_size] |
|  |  |
|  | # second order |
|  | y\_second\_order = 0.5 \* tf.subtract(summed\_features\_emb\_square, squared\_features\_emb\_summed) |
|  |  |
|  |  |
|  | # ----------- Deep Component ------------ |
|  | y\_deep = tf.reshape(embeddings\_origin, shape=[-1, config.field\_size \* config.embedding\_size]) # [None, field\_size \* embedding\_size] |
|  | for i in range(0, len(deep\_layers)): |
|  | y\_deep = tf.add(tf.matmul(y\_deep, weights['layer\_%d' % i]), weights['bias\_%d' % i]) |
|  | y\_deep = config.deep\_layers\_activation(y\_deep) |
|  |  |
|  | # ----------- output ----------- |
|  | concat\_input = tf.concat([y\_first\_order, y\_second\_order, y\_deep], axis=1) |
|  | out = tf.add(tf.matmul(concat\_input, weights['concat\_projection']), weights['concat\_bias']) |
|  | out = tf.nn.sigmoid(out) |
|  |  |
|  | config.loss = "logloss" |
|  | config.l2\_reg = 0.1 |
|  | config.learning\_rate = 0.1 |
|  |  |
|  | # loss |
|  | if config.loss == "logloss": |
|  | loss = tf.losses.log\_loss(label, out) |
|  | elif config.loss == "mse": |
|  | loss = tf.losses.mean\_squared\_error(label, out) |
|  |  |
|  | # l2 |
|  | if config.l2\_reg > 0: |
|  | loss += tf.contrib.layers.l2\_regularizer(config.l2\_reg)(weights['concat\_projection']) |
|  | for i in range(len(deep\_layers)): |
|  | loss += tf.contrib.layers.l2\_regularizer(config.l2\_reg)(weights['layer\_%d' % i]) |
|  |  |
|  | # optimizer |
|  | optimizer = tf.train.AdamOptimizer(learning\_rate=config.learning\_rate, beta1=0.9, beta2=0.999, epsilon=1e-8).minimize(loss) |
|  |  |
|  | ################################## |
|  | # 5. 训练 |
|  | ################################## |
|  |  |
|  | # init session |
|  | sess = tf.Session(graph=tf.get\_default\_graph()) |
|  | sess.run(tf.global\_variables\_initializer()) |
|  |  |
|  | # train |
|  | feed\_dict = { |
|  | feat\_index: Xi\_train, |
|  | feat\_value: Xv\_train, |
|  | label: np.array(y).reshape((-1,1)) |
|  | } |
|  |  |
|  |  |
|  | for epoch in range(config.epochs): |
|  | train\_loss,opt = sess.run((loss, optimizer), feed\_dict=feed\_dict) |
|  | print("epoch: {0}, train loss: {1:.6f}".format(epoch, train\_loss)) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | ################################## |
|  | # 6. 预测 |
|  | ################################## |
|  | dummy\_y = [1] \* len(Xi\_test) |
|  | feed\_dict\_test = { |
|  | feat\_index: Xi\_test, |
|  | feat\_value: Xv\_test, |
|  | label: np.array(dummy\_y).reshape((-1,1)) |
|  | } |
|  |  |
|  | prediction = sess.run(out, feed\_dict=feed\_dict\_test) |
|  |  |
|  | sub = pd.DataFrame({"id":ids, "pred":np.squeeze(prediction)}) |
|  | print("prediction:") |
|  | print(sub) |

