**用TensorFlow实现支持多值、稀疏、共享权重的DeepFM**

**缘起**

DeepFM不算什么新技术了，用TensorFlow实现DeepFM也有开源实现，那我为什么要炒这个冷饭，重复造轮子？

用Google搜索“TensorFlow+DeepFM”，一般都能搜索到“[**ChenglongChen/tensorflow-DeepFM**](https://link.zhihu.com/?target=https%3A//github.com/ChenglongChen/tensorflow-DeepFM)”和“[**lambdaJi的TensorFlow Estimator of DeepFM**](https://zhuanlan.zhihu.com/p/33699909)”这二位的实现。二位不仅用TensorFlow实现了DeepFM，还在Criteo数据集上，给出了完整的训练、测试的代码，的确给了我很大的启发，在这里要表示感谢。

但是，同样是由于二位的实现都是根据Criteo简单数据集的，使他们的代码，如果移植到实际的推荐系统中，存在一定困难。比如：

**稀疏要求**。尽管criteo的原始数据集是排零存储的，但是以上的两个实现，都是用稠密矩阵来表示输入，将0又都补了回来。这种做法，在criteo这种只有39列的简单数据集上是可行的，但是实际系统中，特征数量以千、万计，这种稀疏转稠密的方式是不可取的。

**一列多值的要求**。Criteo数据集有13列numeric特征+26列categorical特征，所有列都只有一个值。但是，在实际系统中，一个field下往往有多个<feature:value>对。比如，我们用三个field来描述一个用户的手机 使用习惯，“近xxx天活跃app”+“近xxx天新安装app”+“近xxx天卸载app”。每个field下，再有“微信:0.9，微博:0.5，淘宝:0.3，……”等一系列的feature和它们的数值。

这个要求固然可以通过，去除field这个“特征单位”，只针对一个个独立的feature来建模。但是，这样一来，既凭空增加了模型的规模，又破坏模型的“层次化”与“模块化”，使代码不易扩展与维护。

**权值共享的要求**。Criteo数据集经过脱敏感处理，我们无法知道每列的具体含义，自然也就没有列与列之间共享权重的需求，以上提到的两个实现也就只用一整块稠密矩阵来建模embedding矩阵。

但是，以上面提到的“近xxx天活跃app”+“近xxx天新安装app”+“近xxx天卸载app”这三个field为例，这些 field中的feature都来源于同一个”app字典”。如果不做权重共享，

* 每个field都使用独立的embedding矩阵来映射app向量，整个模型需要优化的变量是共享权重模型的3倍，既耗费了更多的计算资源，也容易导致过拟合。
* 每个field的稀疏程度是不一样的，同一个app，在“活跃列表”中出现得更频繁，其embedding向量就有更多的训练机会，而在“卸载列表”中较少出现，其embedding向量得不到足够训练，恐怕最后与随机初始化无异。

因此，在实际系统中，“共享权重”是必须的，

* 减小优化变量的数目，既节省计算资源，又减轻“过拟合”风险
* 同一个embedding矩阵，为多个field提供映射向量，类似于“多任务学习”，使每个embedding向量得到更多的训练机会，同时也要满足多个field的需求（比如同一个app的向量，既要体现‘经常使用它’对y的影响，也要体现‘卸载它’对y值的影响），也降低了“过拟合”的风险。

正因为在目前我能够找到的基于TensorFlow实现的DeepFM中，没有一个能够满足以上**“稀疏”、“多值”、“共享权重”**这三个要求的，所以，我自己动手实现了一个，代码见我的[**github**](https://link.zhihu.com/?target=https%3A//github.com/stasi009/Recommend-Estimators/blob/master/deepfm.py)。接下来，我简单讲解一下我的代码。

**数据预处理**

我依然用criteo数据集来做演示之用。为了演示“一列多值”和“稀疏”，我把criteo中的特征分为两个field，所有数值特征I1~I13归为numeric field，所有类别特征C1~C26归为categorical field。

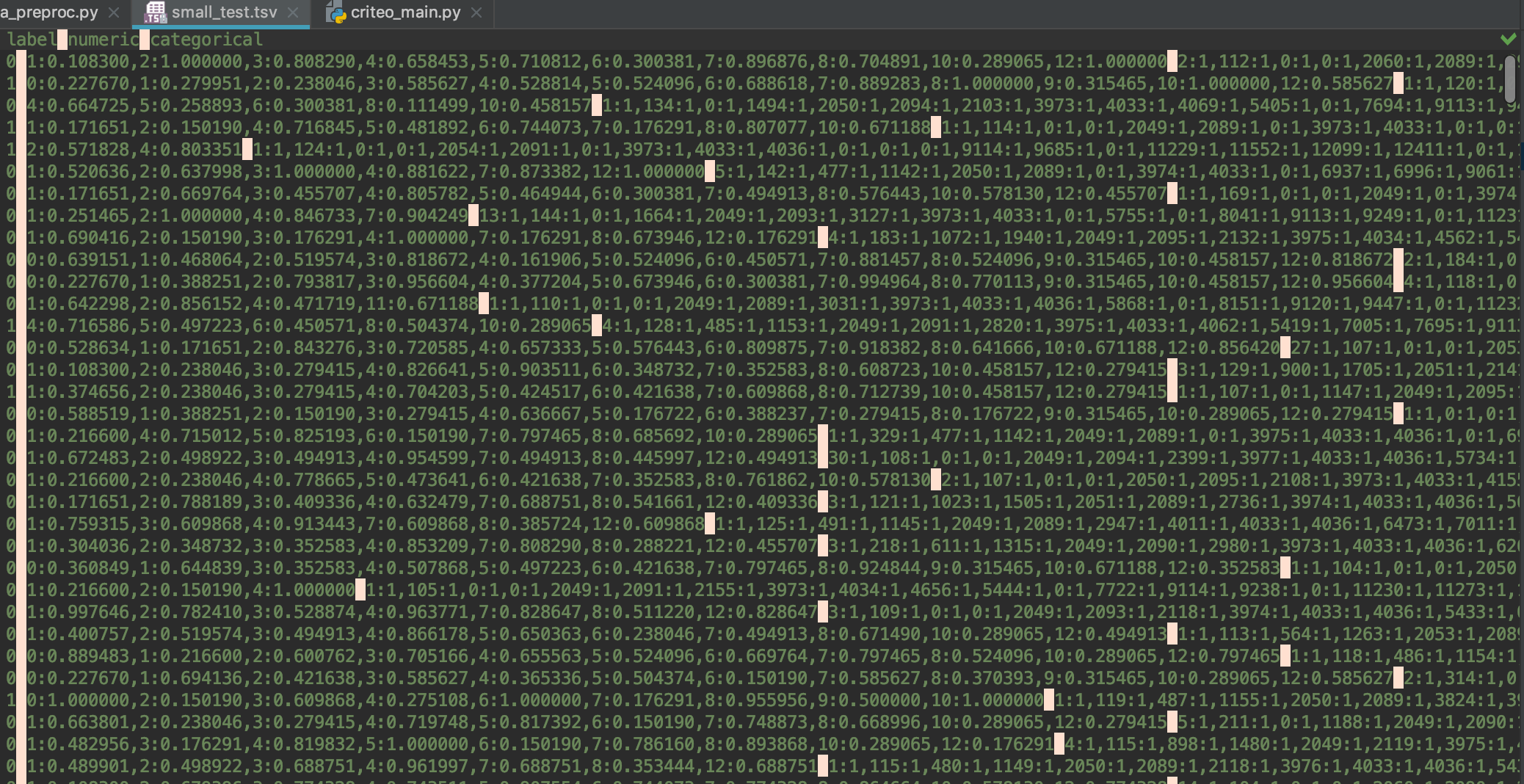
需要特别指出的是：

* 这种处理方法，不是为了提高criteo数据集上的模型性能，只是为了模拟实际系统中将会遇到的“一列多值”和“稀疏”数据集。接下来会看到，DeepFM中，FM中的二阶交叉，不会受拆分成两个field的影响。受影响的主要是Deep侧的输入层，详情见”DNN预测部分”一节 。
* 另外，criteo数据集无法演示“权重共享”的功能。

对criteo中数值特征与类别特征，都是最常规的预处理，不是这次演示的重点

* 数值特征，因为多数表示"次数"，因此先做了一个log变化，减弱长尾数据的影响，再做了一个min/max scaling，毕竟底层还是线性算法，要排除特征间不同scale的影响。注意，**千万不能做“zero mean, unit variance”的standardize，因为那样会破坏数据的稀疏性**。
* 类别特征，剔除了一些生僻的tag，建立字典，将原始数据中的字符串tag转化为整数的index

预处理的代码见[criteo\_data\_preproc.py](https://link.zhihu.com/?target=https%3A//github.com/stasi009/Recommend-Estimators/blob/master/criteo_data_preproc.py)，处理好的数据文件如下所示，图中的亮块是列分隔符。可以看到，每列是由多个tag\_index:value“键值对”组成的，而**不同行中“键值对”个数互不同，而value绝没有0，实现排零、稀疏存储**。



输入数据

**input\_fn**

为了配合TensorFlow Estimator，我们需要定义input\_fn来读取上图所示的数据。

看似简单的任务，实现起来，却很花费了我一番功夫：

* 网上能够搜到的TensorFlow读文本文件的代码，都是读“每列只有一个值的csv”这样规则的数据格式。但是，上图所示的数据，**却非常不规则，每行先是由“\t”分隔，第列中再由“,”分隔成数目不同的“键值对”，每个‘键值对’再由“:”分隔**。
* 我希望提供给model稀疏矩阵，方便model中排零计算，提升效率。

最终，解析一行文本的代码如下。

**def** **\_decode\_tsv**(line):

columns **=** tf**.**decode\_csv(line, record\_defaults**=**DEFAULT\_VALUES, field\_delim**=**'\t')

y **=** columns[0]

feat\_columns **=** dict(zip((t[0] **for** t **in** COLUMNS\_MAX\_TOKENS), columns[1:]))

X **=** {}

**for** colname, max\_tokens **in** COLUMNS\_MAX\_TOKENS:

*# 调用string\_split时，第一个参数必须是一个list，所以要把columns[colname]放在[]中*

*# 这时每个kv还是'k:v'这样的字符串*

kvpairs **=** tf**.**string\_split([feat\_columns[colname]], ',')**.**values[:max\_tokens]

*# k,v已经拆开, kvpairs是一个SparseTensor，因为每个kvpair格式相同，都是"k:v"*

*# 既不会出现"k"，也不会出现"k:v1:v2:v3:..."*

*# 所以，这时的kvpairs实际上是一个满阵*

kvpairs **=** tf**.**string\_split(kvpairs, ':')

*# kvpairs是一个[n\_valid\_pairs,2]矩阵*

kvpairs **=** tf**.**reshape(kvpairs**.**values, kvpairs**.**dense\_shape)

feat\_ids, feat\_vals **=** tf**.**split(kvpairs, num\_or\_size\_splits**=**2, axis**=**1)

feat\_ids **=** tf**.**string\_to\_number(feat\_ids, out\_type**=**tf**.**int32)

feat\_vals **=** tf**.**string\_to\_number(feat\_vals, out\_type**=**tf**.**float32)

*# 不能调用squeeze, squeeze的限制太多, 当原始矩阵有1行或0行时，squeeze都会报错*

X[colname **+** "\_ids"] **=** tf**.**reshape(feat\_ids, shape**=**[**-**1])

X[colname **+** "\_values"] **=** tf**.**reshape(feat\_vals, shape**=**[**-**1])

**return** X, y

然后，将整个文件转化成TensorFlow Dataset的代码如下所示。每一个field“xxx”在dataset中将由两个SparseTensor表示，“xxx\_ids”表示sparse ids，“xxx\_values”表示sparse values。

**def** **input\_fn**(data\_file, n\_repeat, batch\_size, batches\_per\_shuffle):

*# ----------- prepare padding*

pad\_shapes **=** {}

pad\_values **=** {}

**for** c, max\_tokens **in** COLUMNS\_MAX\_TOKENS:

pad\_shapes[c **+** "\_ids"] **=** tf**.**TensorShape([max\_tokens])

pad\_shapes[c **+** "\_values"] **=** tf**.**TensorShape([max\_tokens])

pad\_values[c **+** "\_ids"] **=** **-**1 *# 0 is still valid token-id, -1 for padding*

pad\_values[c **+** "\_values"] **=** 0.0

*# no need to pad labels*

pad\_shapes **=** (pad\_shapes, tf**.**TensorShape([]))

pad\_values **=** (pad\_values, 0)

*# ----------- define reading ops*

dataset **=** tf**.**data**.**TextLineDataset(data\_file)**.**skip(1) *# skip the header*

dataset **=** dataset**.**map(\_decode\_tsv, num\_parallel\_calls**=**4)

**if** batches\_per\_shuffle **>** 0:

dataset **=** dataset**.**shuffle(batches\_per\_shuffle **\*** batch\_size)

dataset **=** dataset**.**repeat(n\_repeat)

dataset **=** dataset**.**padded\_batch(batch\_size**=**batch\_size,

padded\_shapes**=**pad\_shapes,

padding\_values**=**pad\_values)

iterator **=** dataset**.**make\_one\_shot\_iterator()

dense\_Xs, ys **=** iterator**.**get\_next()

*# ----------- convert dense to sparse*

sparse\_Xs **=** {}

**for** c, \_ **in** COLUMNS\_MAX\_TOKENS:

**for** suffix **in** ["ids", "values"]:

k **=** "{}\_{}"**.**format(c, suffix)

sparse\_Xs[k] **=** tf\_utils**.**to\_sparse\_input\_and\_drop\_ignore\_values(dense\_Xs[k])

*# ----------- return*

**return** sparse\_Xs, ys

其中也不得不调用*padded\_batch*补齐，这一步也将稀疏格式转化成了稠密格式，不过只是在一个batch(batch\_size=128已经算很大了)中临时稠密一下，很快就又通过调用to\_sparse\_input\_and\_drop\_ignore\_values这个函数重新转化成稀疏格式了。to\_sparse\_input\_and\_drop\_ignore\_values实际上是从[feature\_column.py](https://link.zhihu.com/?target=https%3A//github.com/tensorflow/tensorflow/blob/r1.11/tensorflow/python/feature_column/feature_column.py)这个module中的\_to\_sparse\_input\_and\_drop\_ignore\_values函数拷贝而来，因为原函数不是public的，无法在featurecolumn.py以外调用，所以我将它的代码拷贝到tf\_utils.py中。

**建立共享权重**

重申几个概念。比如我们的特征集中包括active\_pkgs（app活跃情况）、install\_pkgs（app安装情况）、uninstall\_pkgs（app卸载情况）。每列包含的内容是一系列feature和其数值，比如qq:0.1, weixin:0.9, taobao:1.1, ……。这些feature都来源于同一份名为package的字典

* field就是active\_pkgs、install\_pkgs、uninstall\_pkgs这些大类，是DataFrame中的每一列
* feature就是每个field下包含的具体内容，一个field下允许存在多个feature，比如前面提到的qq, weixin, taobao这样的app名称。
* vocabulary对应例子中的“package字典”。不同field下的feature可以来自同一个vocabulary，即若干field共享vocabulary

建立共享权重的代码如下所示：

* 一个vocab对应两个embedding矩阵，一个对应FM中的线性部分的权重，另一个对应FM与DNN共享的隐向量（用于二阶与高阶交叉）。
* 所有embedding矩阵，以”字典名”存入dict。**不同field只要指定相同的“字典名”，就可以共享同一套embedding矩阵**。

**class** **EmbeddingTable**:

**def** **\_\_init\_\_**(self):

self**.**\_weights **=** {}

**def** **add\_weights**(self, vocab\_name, vocab\_size, embed\_dim):

"""

:param vocab\_name: 一个field拥有两个权重矩阵，一个用于线性连接，另一个用于非线性（二阶或更高阶交叉）连接

:param vocab\_size: 字典总长度

:param embed\_dim: 二阶权重矩阵shape=[vocab\_size, order2dim]，映射成的embedding

既用于接入DNN的第一屋，也是用于FM二阶交互的隐向量

:return: None

"""

linear\_weight **=** tf**.**get\_variable(name**=**'{}\_linear\_weight'**.**format(vocab\_name),

shape**=**[vocab\_size, 1],

initializer**=**tf**.**glorot\_normal\_initializer(),

dtype**=**tf**.**float32)

*# 二阶（FM）与高阶（DNN）的特征交互，共享embedding矩阵*

embed\_weight **=** tf**.**get\_variable(name**=**'{}\_embed\_weight'**.**format(vocab\_name),

shape**=**[vocab\_size, embed\_dim],

initializer**=**tf**.**glorot\_normal\_initializer(),

dtype**=**tf**.**float32)

self**.**\_weights[vocab\_name] **=** (linear\_weight, embed\_weight)

**def** **get\_linear\_weights**(self, vocab\_name): **return** self**.**\_weights[vocab\_name][0]

**def** **get\_embed\_weights**(self, vocab\_name): **return** self**.**\_weights[vocab\_name][1]

**def** **build\_embedding\_table**(params):

embed\_dim **=** params['embed\_dim'] *# 必须有统一的embedding长度*

embedding\_table **=** EmbeddingTable()

**for** vocab\_name, vocab\_size **in** params['vocab\_sizes']**.**items():

embedding\_table**.**add\_weights(vocab\_name**=**vocab\_name, vocab\_size**=**vocab\_size, embed\_dim**=**embed\_dim)

**return** embedding\_table

**线性预测部分**

**def** **output\_logits\_from\_linear**(features, embedding\_table, params):

field2vocab\_mapping **=** params['field\_vocab\_mapping']

combiner **=** params**.**get('multi\_embed\_combiner', 'sum')

fields\_outputs **=** []

*# 当前field下有一系列的<tag:value>对，每个tag对应一个bias（待优化），*

*# 将所有tag对应的bias，按照其value进行加权平均，得到这个field对应的bias*

**for** fieldname, vocabname **in** field2vocab\_mapping**.**items():

sp\_ids **=** features[fieldname **+** "\_ids"]

sp\_values **=** features[fieldname **+** "\_values"]

linear\_weights **=** embedding\_table**.**get\_linear\_weights(vocab\_name**=**vocabname)

*# weights: [vocab\_size,1]*

*# sp\_ids: [batch\_size, max\_tags\_per\_example]*

*# sp\_weights: [batch\_size, max\_tags\_per\_example]*

*# output: [batch\_size, 1]*

output **=** embedding\_ops**.**safe\_embedding\_lookup\_sparse(linear\_weights, sp\_ids, sp\_values,

combiner**=**combiner,

name**=**'{}\_linear\_output'**.**format(fieldname))

fields\_outputs**.**append(output)

*# 因为不同field可以共享同一个vocab的linear weight，所以将各个field的output相加，会损失大量的信息*

*# 因此，所有field对应的output拼接起来，反正每个field的output都是[batch\_size,1]，拼接起来，并不占多少空间*

*# whole\_linear\_output: [batch\_size, total\_fields]*

whole\_linear\_output **=** tf**.**concat(fields\_outputs, axis**=**1)

tf**.**logging**.**info("linear output, shape={}"**.**format(whole\_linear\_output**.**shape))

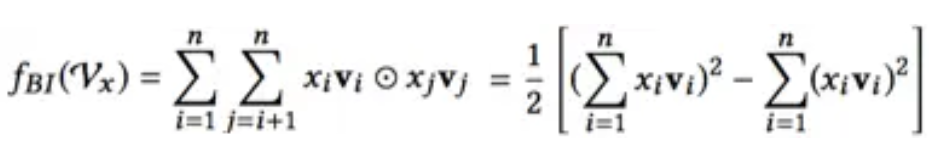
*# 再映射到final logits（二分类，也是[batch\_size,1]）*

*# 这时，就不要用任何activation了，特别是ReLU*

**return** tf**.**layers**.**dense(whole\_linear\_output, units**=**1, use\_bias**=True**, activation**=None**)

**二阶交互预测部分**

二阶交互部分与DeepFM论文中稍有不同，而是使用了《Neural Factorization Machines for Sparse Predictive Analytics》中Bi-Interaction的公式。这也是网上实现的通用做法。



而我的实现与上边公式最大的不同，就是**不再只有一个embedding矩阵V**，而是每个feature根据自己所在的field，再根据超参指定的field与vocabulary的映射关系，找到自己对应的embedding矩阵。某个field对应的embedding矩阵有可能是与另外一个field共享的。

另外， xivi 实现了**稀疏矩阵相乘**，基于embedding\_ops.**safe\_embedding\_lookup\_sparse**实现。

**def** **output\_logits\_from\_bi\_interaction**(features, embedding\_table, params):

field2vocab\_mapping **=** params['field\_vocab\_mapping']

*# 论文上的公式就是要求sum，而且我也试过mean和sqrtn，都比用mean要差上很多*

*# 但是，这种情况，仅仅是针对criteo数据的，还是理论上就必须用sum，而不能用mean和sqrtn*

*# 我还不太确定，所以保留一个接口能指定其他combiner的方法*

combiner **=** params**.**get('multi\_embed\_combiner', 'sum')

*# 见《Neural Factorization Machines for Sparse Predictive Analytics》论文的公式(4)*

fields\_embeddings **=** []

fields\_squared\_embeddings **=** []

**for** fieldname, vocabname **in** field2vocab\_mapping**.**items():

sp\_ids **=** features[fieldname **+** "\_ids"]

sp\_values **=** features[fieldname **+** "\_values"]

*# --------- embedding*

embed\_weights **=** embedding\_table**.**get\_embed\_weights(vocabname)

*# embedding: [batch\_size, embed\_dim]*

embedding **=** embedding\_ops**.**safe\_embedding\_lookup\_sparse(embed\_weights, sp\_ids, sp\_values,

combiner**=**combiner,

name**=**'{}\_embedding'**.**format(fieldname))

fields\_embeddings**.**append(embedding)

*# --------- square of embedding*

squared\_emb\_weights **=** tf**.**square(embed\_weights)

squared\_sp\_values **=** tf**.**SparseTensor(indices**=**sp\_values**.**indices,

values**=**tf**.**square(sp\_values**.**values),

dense\_shape**=**sp\_values**.**dense\_shape)

*# squared\_embedding: [batch\_size, embed\_dim]*

squared\_embedding **=** embedding\_ops**.**safe\_embedding\_lookup\_sparse(squared\_emb\_weights, sp\_ids, squared\_sp\_values,

combiner**=**combiner,

name**=**'{}\_squared\_embedding'**.**format(fieldname))

fields\_squared\_embeddings**.**append(squared\_embedding)

*# calculate bi-interaction*

sum\_embedding\_then\_square **=** tf**.**square(tf**.**add\_n(fields\_embeddings)) *# [batch\_size, embed\_dim]*

square\_embedding\_then\_sum **=** tf**.**add\_n(fields\_squared\_embeddings) *# [batch\_size, embed\_dim]*

bi\_interaction **=** 0.5 **\*** (sum\_embedding\_then\_square **-** square\_embedding\_then\_sum) *# [batch\_size, embed\_dim]*

tf**.**logging**.**info("bi-interaction, shape={}"**.**format(bi\_interaction**.**shape))

*# calculate logits*

logits **=** tf**.**layers**.**dense(bi\_interaction, units**=**1, use\_bias**=True**, activation**=None**)

*# 因为FM与DNN共享embedding，所以除了logits，还返回各field的embedding，方便搭建DNN*

**return** logits, fields\_embeddings

**DNN预测部分**

再次声明，**将criteo中原来的39列，拆分成2个field，并不是为了提升预测性能，只是为了模拟实际场景**。导致的后果就是，Deep侧第一层的输入由原来的[batch\_size, 39\*embed\_dim]变成了[batch\_size, 2\*embed\_dim]，使Deep侧交叉不足。

尽管在criteo数据集上，deep侧的输入由**feature\_size**\*embed\_dim变成了**field\_size**\*embed\_dim，限制了交叉能力。但是，在实际系统中，**field\_size已经是成千上万了，而每个field下的feature又是成千上万**，而且，因为embedding是稠密的，没有稀疏优化的可能性。因此，在接入deep侧之前，每个field内部先做一层pooling，将deep侧输入由**feature\_size\*embed\_dim压缩成field\_size\*embed\_dim**，对于大规模机器学习，是十分必要的。

DNN的代码如下所示。可以看到，其中没有加入L1/L2 regularization，这是模仿TensorFlow自带的Wide & Deep实现DNNLinearCombinedClassifier的写法。L1/L2正则将通过设置optimizer的参数来实现。

**def** **output\_logits\_from\_dnn**(fields\_embeddings, params, is\_training):

dropout\_rate **=** params['dropout\_rate']

do\_batch\_norm **=** params['batch\_norm']

X **=** tf**.**concat(fields\_embeddings, axis**=**1)

tf**.**logging**.**info("initial input to DNN, shape={}"**.**format(X**.**shape))

**for** idx, n\_units **in** enumerate(params['hidden\_units'], start**=**1):

X **=** tf**.**layers**.**dense(X, units**=**n\_units, activation**=**tf**.**nn**.**relu)

tf**.**logging**.**info("layer[{}] output shape={}"**.**format(idx, X**.**shape))

X **=** tf**.**layers**.**dropout(inputs**=**X, rate**=**dropout\_rate, training**=**is\_training)

**if** is\_training:

tf**.**logging**.**info("layer[{}] dropout {}"**.**format(idx, dropout\_rate))

**if** do\_batch\_norm:

*# BatchNormalization的调用、参数，是从DNNLinearCombinedClassifier源码中拷贝过来的*

batch\_norm\_layer **=** normalization**.**BatchNormalization(momentum**=**0.999, trainable**=True**,

name**=**'batchnorm\_{}'**.**format(idx))

X **=** batch\_norm\_layer(X, training**=**is\_training)

**if** is\_training:

tf**.**logging**.**info("layer[{}] batch-normalize"**.**format(idx))

*# connect to final logits, [batch\_size,1]*

**return** tf**.**layers**.**dense(X, units**=**1, use\_bias**=True**, activation**=None**)

**model\_fn**

前面的代码完成了“线性预测”+“二次交叉预测”+“深度预测”，则model\_fn的实现就非常简单了，只不过将三个部分得到的logits相加就可以了。

**def** **model\_fn**(features, labels, mode, params):

**for** featname, featvalues **in** features**.**items():

**if** **not** isinstance(featvalues, tf**.**SparseTensor):

**raise** **TypeError**("feature[{}] isn't SparseTensor"**.**format(featname))

*# ============= build the graph*

embedding\_table **=** build\_embedding\_table(params)

linear\_logits **=** output\_logits\_from\_linear(features, embedding\_table, params)

bi\_interact\_logits, fields\_embeddings **=** output\_logits\_from\_bi\_interaction(features, embedding\_table, params)

dnn\_logits **=** output\_logits\_from\_dnn(fields\_embeddings, params, (mode **==** tf**.**estimator**.**ModeKeys**.**TRAIN))

general\_bias **=** tf**.**get\_variable(name**=**'general\_bias', shape**=**[1], initializer**=**tf**.**constant\_initializer(0.0))

logits **=** linear\_logits **+** bi\_interact\_logits **+** dnn\_logits

logits **=** tf**.**nn**.**bias\_add(logits, general\_bias) *# bias\_add，获取broadcasting的便利*

*# reshape [batch\_size,1] to [batch\_size], to match the shape of 'labels'*

logits **=** tf**.**reshape(logits, shape**=**[**-**1])

probabilities **=** tf**.**sigmoid(logits)

*# ============= predict spec*

**if** mode **==** tf**.**estimator**.**ModeKeys**.**PREDICT:

**return** tf**.**estimator**.**EstimatorSpec(

mode**=**mode,

predictions**=**{'probabilities': probabilities})

*# ============= evaluate spec*

*# STUPID TENSORFLOW CANNOT AUTO-CAST THE LABELS FOR ME*

loss **=** tf**.**reduce\_mean(tf**.**nn**.**sigmoid\_cross\_entropy\_with\_logits(logits**=**logits, labels**=**tf**.**cast(labels, tf**.**float32)))

eval\_metric\_ops **=** {'auc': tf**.**metrics**.**auc(labels, probabilities)}

**if** mode **==** tf**.**estimator**.**ModeKeys**.**EVAL:

**return** tf**.**estimator**.**EstimatorSpec(

mode**=**mode,

loss**=**loss,

eval\_metric\_ops**=**eval\_metric\_ops)

*# ============= train spec*

**assert** mode **==** tf**.**estimator**.**ModeKeys**.**TRAIN

train\_op **=** params['optimizer']**.**minimize(loss, global\_step**=**tf**.**train**.**get\_global\_step())

**return** tf**.**estimator**.**EstimatorSpec(mode,

loss**=**loss,

train\_op**=**train\_op,

eval\_metric\_ops**=**eval\_metric\_ops)

**训练与评估**

完成了model\_fn之后，拜TensorFlow Estimator框架所赐，训练与评估变得非常简单，设定超参数之后（注意在指定optimizer时设置了L1/L2的正则权重），调用tf.estimator.train\_and\_evaluate即可。

**def** **get\_hparams**():

vocab\_sizes **=** {

'numeric': 13,

*# there are totally 14738 categorical tags occur >= 200*

*# since 0 is reserved for OOV, so total vocab\_size=14739*

'categorical': 14739

}

optimizer **=** tf**.**train**.**ProximalAdagradOptimizer(

learning\_rate**=**0.01,

l1\_regularization\_strength**=**0.001,

l2\_regularization\_strength**=**0.001)

**return** {

'embed\_dim': 128,

'vocab\_sizes': vocab\_sizes,

*# 在这个case中，没有多个field共享同一个vocab的情况，而且field\_name和vocab\_name相同*

'field\_vocab\_mapping': {'numeric': 'numeric', 'categorical': 'categorical'},

'dropout\_rate': 0.3,

'batch\_norm': **False**,

'hidden\_units': [64, 32],

'optimizer': optimizer

}

**if** \_\_name\_\_ **==** "\_\_main\_\_":

tf**.**logging**.**set\_verbosity(tf**.**logging**.**INFO)

tf**.**set\_random\_seed(999)

hparams **=** get\_hparams()

deepfm **=** tf**.**estimator**.**Estimator(model\_fn**=**model\_fn,

model\_dir**=**'models/criteo',

params**=**hparams)

train\_spec **=** tf**.**estimator**.**TrainSpec(input\_fn**=lambda**: input\_fn(data\_file**=**'dataset/criteo/whole\_train.tsv',

n\_repeat**=**10,

batch\_size**=**128,

batches\_per\_shuffle**=**10))

eval\_spec **=** tf**.**estimator**.**EvalSpec(input\_fn**=lambda**: input\_fn(data\_file**=**'dataset/criteo/whole\_test.tsv',

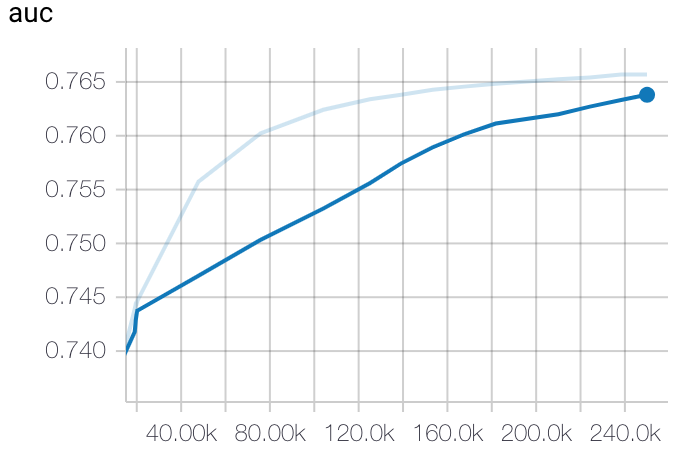
n\_repeat**=**1,

batch\_size**=**128,

batches\_per\_shuffle**=-**1))

tf**.**estimator**.**train\_and\_evaluate(deepfm, train\_spec, eval\_spec)

测试集上的部分结果所下所示，测试集上的AUC在0.765左右，没有Kaggle solution上0.8+的AUC高。正如前文所说的，将原来criteo数据集中的39列拆分成2个field，只是为了演示“一列多值”、“稀疏”的DeepFM实现，但限制了Deep侧的交叉能力，对最终模型的性能造成一定负面影响。不过，仍然证明，文中展示的DeepFM实现是正确的。



**小结**

本文展示了我写的一套基于TensorFlow的DeepFM的实现。重点阐述了**“一列多值”、“稀疏”、“权重共享”**在实际推荐系统中的重要性，和我是如何在DeepFM中实现以上需求的。欢迎各位看官指正。