目录

[一．Wide And Deep 模型定义 2](#_Toc4648)

[1.获取输入 2](#_Toc28415)

[2.定义 feature columns 2](#_Toc8798)

[3.定义模型 3](#_Toc26893)

[3.1分类模型 3](#_Toc8202)

[3.2回归模型 3](#_Toc4516)

[4.训练评测 4](#_Toc4625)

[5.输出 5](#_Toc1607)

[二．模型参数详解 5](#_Toc10654)

[1.基础参数 5](#_Toc24163)

[2.线性模型相关参数 6](#_Toc18346)

[3.DNN 模型相关参数 6](#_Toc2368)

[三．特征 7](#_Toc20400)

[1.sparse column from keys 12](#_Toc4219)

[2.sparse column from vocabulary file 13](#_Toc12472)

[3.sparse column with hash bucket 15](#_Toc2898)

[4.integerized sparse column 15](#_Toc24572)

[5.crossed column 16](#_Toc12617)

[6.real valued column 18](#_Toc6516)

[7.bucketized column 20](#_Toc12012)

[8.embedding column 20](#_Toc7693)

[9.其他 feature columns 24](#_Toc13759)

[四．训练实现 25](#_Toc9308)

[1.内部函数的调用过程 25](#_Toc18589)

[2.模型训练的步骤 28](#_Toc6144)

[2.1从 params 获取参数，定义模型优化函数 28](#_Toc4113)

[2.2构建 DNN 模型 28](#_Toc25885)

[2.3构建线性模型 31](#_Toc7790)

[2.4combine 32](#_Toc14476)

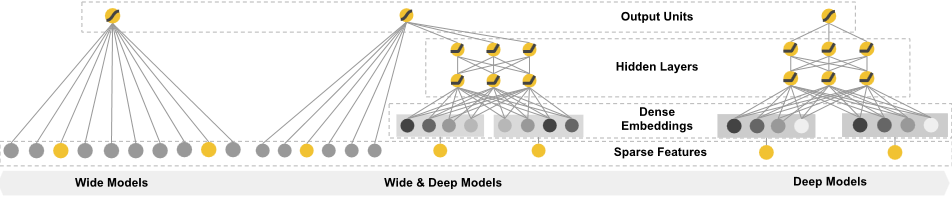
[2.5参数更新 33](#_Toc2481)

[五．Wide And Deep 模型应用 39](#_Toc8886)

**整理自：<http://geek.csdn.net/news/detail/235465>**

Wide and deep 模型是 TensorFlow 在 2016 年 6 月左右发布的一类用于分类和回归的模型，并应用到了 Google Play 的应用推荐中 [1]。wide and deep 模型的核心思想是结合线性模型的记忆能力（memorization）和 DNN 模型的泛化能力（generalization），在训练过程中同时优化 2 个模型的参数，从而达到整体模型的预测能力最优。

wide 端对应的是线性模型，输入特征可以是连续特征，也可以是稀疏的离散特征，离散特征之间进行交叉后可以构成更高维的离散特征。线性模型训练中通过 L1 正则化，能够很快收敛到有效的特征组合中。deep 端对应的是 DNN 模型，每个特征对应一个低维的实数向量，我们称之为特征的 embedding。DNN 模型通过反向传播调整隐藏层的权重，并且更新特征的 embedding。wide and deep 整个模型的输出是线性模型输出与 DNN 模型输出的叠加。



模型训练采用的是联合训练（joint training），模型的训练误差会同时反馈到线性模型和 DNN 模型中进行参数更新。相比于 ensemble learning 中单个模型进行独立训练，模型的融合仅在最终做预测阶段进行，joint training 中模型的融合是在训练阶段进行的，单个模型的权重更新会受到 wide 端和 deep 端对模型训练误差的共同影响。因此在模型的特征设计阶段，wide 端模型和 deep 端模型只需要分别专注于擅长的方面，wide 端模型通过离散特征的交叉组合进行 memorization，deep 端模型通过特征的 embedding 进行 generalization，这样单个模型的大小和复杂度也能得到控制，而整体模型的性能仍能得到提高。

# 一．Wide And Deep 模型定义

## 1.获取输入

模型的输入是一个 python 的 dataframe。如 tutorial 的实例代码，可以通过 pandas.read\_csv 从 CSV 文件中读入数据构建 data frame。

## 2.定义 feature columns

tf.contrib.layers 中提供了一系列的函数定义不同类型的 feature columns：

* tf.contrib.layers.sparse\_column\_with\_XXX 构建低维离散特征   
  sparse\_feature\_a = sparse\_column\_with\_hash\_bucket(…)   
  sparse\_feature\_b = sparse\_column\_with\_hash\_bucket(…)
* tf.contrib.layers.crossed\_column 构建离散特征的组合   
  sparse\_feature\_a\_x\_sparse\_feature\_b = crossed\_column([sparse\_feature\_a, sparse\_feature\_b], …)
* tf.contrib.layers.real\_valued\_column 构建连续型实数特征   
  real\_feature\_a = real\_valued\_column(…)
* tf.contrib.layers.embedding\_column 构建 embedding 特征   
  sparse\_feature\_a\_emb = embedding\_column(sparse\_id\_column=sparse\_feature\_a, )

## 3.定义模型

### 3.1分类模型

m = tf.contrib.learn.DNNLinearCombinedClassifier(

n\_classes = n\_classes, // 分类数目

weight\_column\_name = weight\_column\_name, // 训练实例的权重

model\_dir = model\_dir, // 模型目录

linear\_feature\_columns = wide\_columns, // 输入线性模型的 feature columns

linear\_optimizer = tf.train.FtrlOptimizer(...), // 线性模型权重更新的 optimizer

dnn\_feature\_columns = deep\_columns, // 输入 DNN 模型的 feature columns

dnn\_hidden\_units=[100, 50]，// DNN 模型的隐藏层单元数目

dnn\_optimizer=tf.train.AdagradOptimizer(...) // DNN 模型权重更新的 optimizer

)

需要指出的是：模型的 model\_dir 同下面会提到的 export 模型的目录是 2 个不同的目录，model\_dir 存放模型的 graph 和 summary 数据，如果 model\_dir 存放了上一次训练的模型数据，训练时会从 model\_dir 恢复上一次训练的模型并在此基础上进行训练。我们用 tensorboard 加载显示的模型数据也是从该目录下生成的。模型 export 的目录则主要是用于 tensorflow server 启动时加载模型的 servable 实例，用于线上预测服务。

### 3.2回归模型

可以如下定义：

m = tf.contrib.learn.DNNLinearCombinedRegressor(

weight\_column\_name = weight\_column\_name,

linear\_feature\_columns = wide\_columns,

linear\_optimizer = tf.train.FtrlOptimizer(...),

dnn\_feature\_columns = deep\_columns,

dnn\_hidden\_units=[100, 50]，

dnn\_optimizer=tf.train.AdagradOptimizer(...)

)

## 4.训练评测

训练模型可以使用 fit 函数：m.fit(input\_fn=input\_fn(df\_train))，评测使用 evaluate 函数：m.evaluate(input\_fn=input\_fn(df\_test))。Input\_fn 函数定义如何从输入的 dataframe 构建特征和标记：

def input\_fn(df)

// tf.constant 构建 constant tensor，df[k].values 是对应 feature column 的值构成的 list

continuous\_cols = {k: tf.constant(df[k].values) for k in CONTINUOUS\_COLUMNS}

// tf.SparseTensor 构建 sparse tensor，SparseTensor 由 indices,values, dense\_shape 三

// 个 dense tensor 构成，indices 中记录非零元素在 sparse tensor 的位置，values 是

// indices 中每个位置的元素的值，dense\_shape 指定 sparse tensor 中每个维度的大小

// 以下代码为每个 category column 构建一个 [df[k].size，1] 的二维的 SparseTensor。

categorical\_cols = {

k: tf.SparseTensor( indices=[[i, 0] for i in range(df[k].size)],

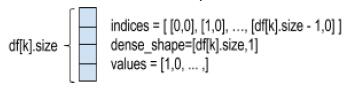
values=df[k].values,

dense\_shape=[df[k].size, 1])

for k in CATEGORICAL\_COLUMNS

}

// 可以用以下示意图来表示以上代码构建的 sparse tensor



// label 是一个 constant tensor，记录每个实例的 label

label = tf.constant(df[LABEL\_COLUMN].values)

// features 是 continuous\_cols 和 categorical\_cols 的 union 构成的 dict

// dict 中每个 entry 的 key 是 feature column 的 name，value 是 feature column 值的 tensor

return features, label

## 5.输出

模型通过 export 输出到一个指定目录，tensorflow serving 从该目录加载模型提供在线预测服务：m.export(export\_dir=export\_dir,input\_fn = export.\_default\_input\_fn   
use\_deprecated\_input\_fn=True,signature\_fn=signature\_fn)   
input\_fn 函数定义生成模型 servable 实例的特征，signature\_fn 函数定义模型输入输出的 signature。   
由于在 TensorFlow1.0 之后 export 已经 deprecate，需要用 export\_savedmodel 来替代，所以本文就不对 export 进行更多讲解，只在文末给出我们是如何使用它的，建议所有使用者以后切换到最新的 API。

# 二．模型参数详解

wide and deep 模型是基于 TF.learn API 来实现的，其源代码实现主要在 tensorflow.contrib.learn.python.learn.estimators 中。以分类模型为例，wide 与 deep 结合的分类模型对应的类是 DNNLinearCombinedClassifier，实现在源文件 dnn\_linear\_combined.py。我们先看看 DNNLinearCombinedClassifier 的初始化函数的完整定义，看构造一个 wide and deep 模型可以输入哪些参数：

def \_\_init\_\_(self, model\_dir=None, n\_classes=2, weight\_column\_name=None, linear\_feature\_columns=None,

linear\_optimizer=None, joint\_linear\_weights=False, dnn\_feature\_columns=None,

dnn\_optimizer=None, dnn\_hidden\_units=None, dnn\_activation\_fn=nn.relu, dnn\_dropout=None,

gradient\_clip\_norm=None, enable\_centered\_bias=False, config=None,

feature\_engineering\_fn=None, embedding\_lr\_multipliers=None):

我们可以将类的构造函数中的参数分为以下几组

## 1.基础参数

model\_dir   
我们训练的模型存放到 model\_dir 指定的目录中。如果我们需要用 tensorboard 来 DEBUG 模型，将 tensorboard 的 logdir 指向该目录即可：tensorboard –logdir=$model\_dir

n\_classes   
分类数。默认是二分类，>2 则进行多分类。

weight\_column\_name   
定义每个训练样本的权重。训练时每个训练样本的训练误差乘以该样本的权重然后用于权重更新梯度的计算。如果需要为每个样本指定权重，input\_fn 返回的 features 里需要包含一个以 weight\_column\_name 为列名的列，该列的长度为训练样本的数目，列中每个元素对应一个样本的权重，数据类型是 float，如以下伪代码：

weight = tf.constant(df[WEIGHT\_COLUMN\_NAME].values, dtype=float32);

features[weight\_column\_name] = weight

config   
指定运行时配置参数

eature\_engineering\_fn   
对输入函数 input\_fn 输出的 (features, label) 进行后处理生成新的 (features』, label』) 然后输入给模型训练函数 model\_fn 使用。

call\_model\_fn():

feature, labels = self.\_feature\_engineering\_fn(feature, labels)

## 2.线性模型相关参数

linear\_feature\_columns   
线性模型的输入特征

linear\_optimizer   
线性模型的优化函数，定义权重的梯度更新算法，默认采用 FTRL。所有默认支持的 linear\_optimizer 和 dnn\_optimizer 可以在 optimizer.py 的 OPTIMIZER\_CLS\_NAMES 变量中找到相关定义。

join\_linear\_weights   
按照代码中的注释，如果 join\_linear\_weights= true，线性模型的权重会存放在一个 tf.Variable 中，可以加快训练，但是 linear\_feature\_columns 中的特征列必须都是 sparse feature column 并且每个 feature column 的 combiner 必须是“sum”。经过自己线下的对比试验，对模型的预测能力似乎没有太大影响，对训练速度有所提升，最终训练模型时我们保持了默认值。

## 3.DNN 模型相关参数

dnn\_feature\_columns   
DNN 模型的输入特征

dnn\_optimizer   
DNN 模型的优化函数，定义各层权重的梯度更新算法，默认采用 Adagrad。

dnn\_hidden\_units   
每个隐藏层的神经元数目

dnn\_activation\_fn   
隐藏层的激活函数，默认采用 RELU

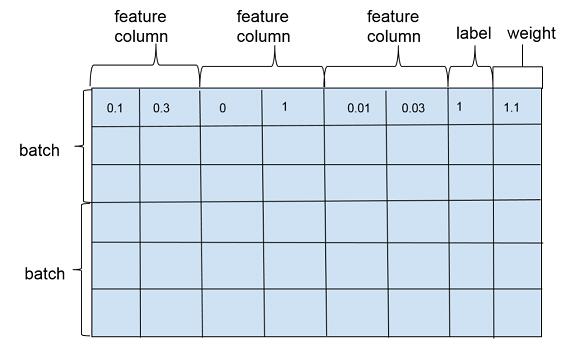
dnn\_dropout   
模型训练中隐藏层单元的 drop\_out 比例

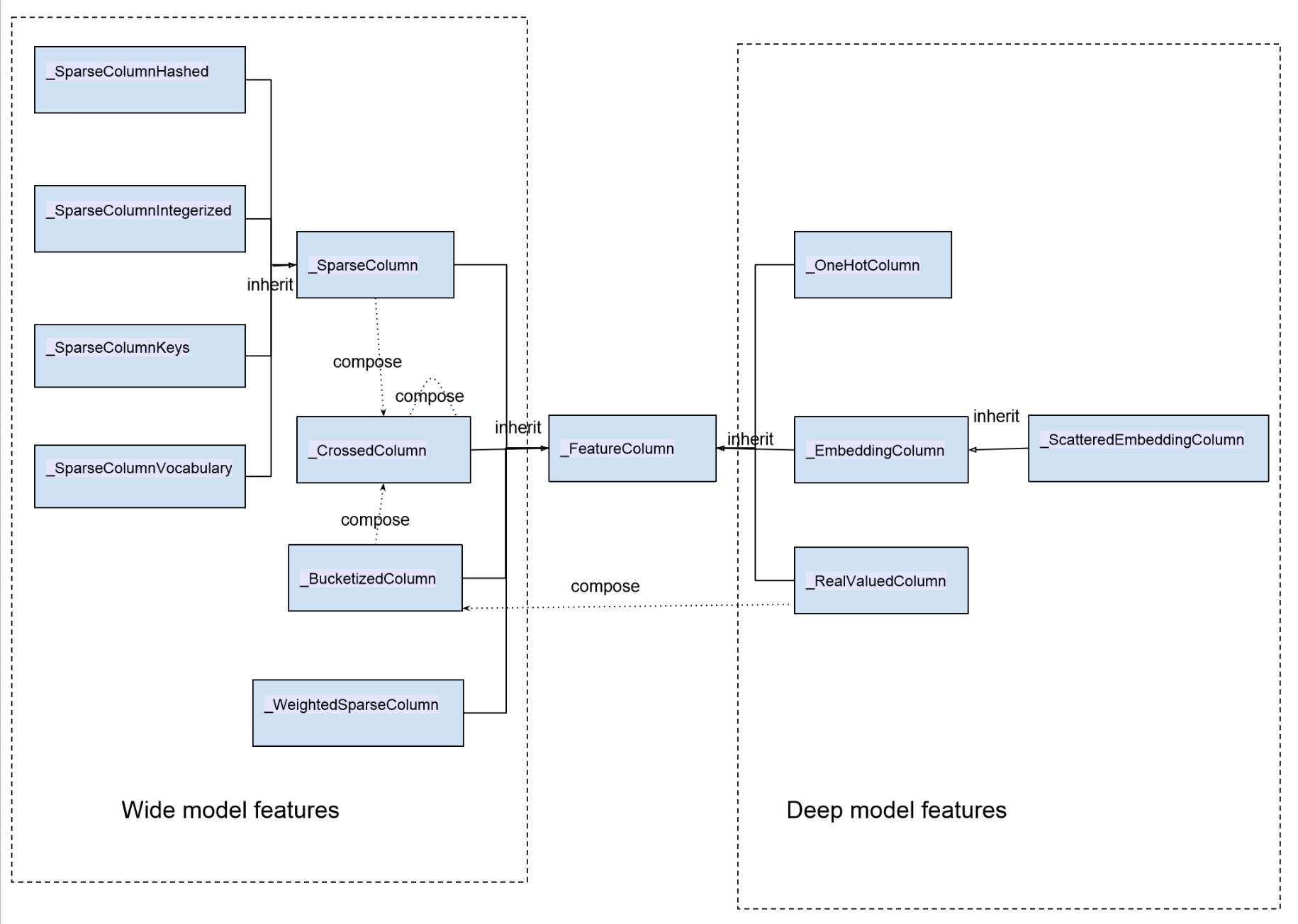
gradient\_clip\_norm   
定义 gradient clipping，对梯度的变化范围做出限制，防止 gradient vanishing 或 gradient explosion。wide and deep 中默认采用 tf.clip\_by\_global\_norm。

embedding\_lr\_multipliers   
embedding\_feature\_column 到 float 的一个 mapping。对指定的 embedding feature column 在计算梯度时乘以一个常数因子，调整梯度的变化速率。

看完模型的构造函数后，我们大概知道 wide 和 deep 端的模型各对应什么样的模型，模型需要输入什么样的参数。为了更深入了解模型，以下我们对 wide and deep 模型的相关代码进行了分析，力求解决如下疑问： (1) 分别用于线性模型和 DNN 模型训练的特征是如何定义的，其内部如何实现；(2) 训练中线性模型和 DNN 模型如何进行联合训练，训练误差如何反馈给 wide 模型和 deep 模型？下面我们重点针对特征和模型训练这两方面进行解读。

# 三．特征

wide and deep 模型训练一般是以多个训练样本作为 1 个批次 (batch) 进行训练，训练样本在行维度上定义，每一行对应一个训练样本实例，包括特征（feature column），标注（label）以及权重（weight），如图 2。特征在列维度上定义，每个特征对应 1 个 feature column，feature column 由在列维度上的 1 个或者若干个张量 (tensor) 组成，tensor 中的每个元素对应一个样本在该 feature column 上某个维度的值。feature column 的定义在可以在源代码的 feature\_column.py 文件中找到，对应类为\_FeatureColumn，该类定义了基本接口，是 wide and deep 模型中所有特征类的抽象父类。

wide and deep 模型中使用的特征包括两大类： 一类是连续型特征，主要用于 deep 模型的训练，包括 real value 类型的特征以及 embedding 类型的特征等；一类是离散型特征，主要用于 wide 模型的训练，包括 sparse 类型的特征以及 cross 类型的特征等。以下是所有特征的一个汇总图

图中类与类的关系除了 inherit（继承）之外，同时我们也标出了特征类之间的构成关系：\_BucketizedColumn 由\_RealValueColumn 通过对连续值域进行分桶构成，\_CrossedColumn 由若干\_SparseColumn 或者\_BucketizedColumn 或者\_CrossedColumn 经过交叉组合构成。图中左边部分特征属于离散型特征，右边部分特征属于连续型特征。

我们在实际使用的时候，通常情况下是调用 TensorFlow 提供的接口来构建特征的。以下是构建各类特征的接口：

sparse\_column\_with\_integerized\_feature() --> \_SparseColumnIntegerized

sparse\_column\_with\_hash\_bucket() --> \_SparseColumnHashed

sparse\_column\_with\_keys() --> \_SparseColumnKeys

sparse\_column\_with\_vocabulary\_file() --> \_SparseColumnVocabulary

weighted\_sparse\_column() --> \_WeightedSparseColumn

one\_hot\_column() --> \_OneHotColumn

embedding\_column() --> \_EmbeddingColumn

shared\_embedding\_columns() --> List[\_EmbeddingColumn]

scattered\_embedding\_column() --> \_ScatteredEmbeddingColumn

real\_valued\_column() --> \_RealValuedColumn

bucketized\_column() -->\_BucketizedColumn

crossed\_column() --> \_CrossedColumn

FeatureColumn 为模型训练定义了几个基本接口用于提取和转换特征，在后面讲解具体 feature 时会有具体描述：

def insert\_transformed\_feature(self, columns\_to\_tensors):   
“”“Apply transformation and inserts it into columns\_to\_tensors.   
FeatureColumn 的特征输出和转换函数。columns\_to\_tensor 是 FeatureColumn 到 tensors 的映射。

def \_to\_dnn\_input\_layer(self, input\_tensor, weight\_collection=None, trainable=True, output\_rank=2):   
“”“Returns a Tensor as an input to the first layer of neural network.”“”   
构建 DNN 的 float tensor 输入，参见后面对 RealValuedColumn 的讲解。

def \_deep\_embedding\_lookup\_arguments(self, input\_tensor):   
“”“Returns arguments to embedding lookup to build an input layer.”“”   
构建 DNN 的 embedding 输入，参见后面对 EmbeddingColumn 的讲解。

def \_wide\_embedding\_lookup\_arguments(self, input\_tensor):   
“”“Returns arguments to look up embeddings for this column.”“”   
构建线性模型的输入，参见后面对 SparseColumn 的讲解。

我们从离散型的特征（sparse 特征）开始分析。离散型特征可以看做由若干键值构成的特征，比如用户的性别。在实际实现中，每一个键值在 sparse column 内部对应一个整数 id。离散特征的基类是\_SparseColumn：

class \_SparseColumn(\_FeatureColumn,

collections.namedtuple("\_SparseColumn",

["column\_name", "is\_integerized",

"bucket\_size", "lookup\_config",

"combiner", "dtype"])):

collections.namedtuple 中的字符串数组是\_SparseColumn 从对应的创建接口函数中接收的输入参数的名称。

def \_\_new\_\_(cls,

column\_name,

is\_integerized=False,

bucket\_size=None,

lookup\_config=None,

combiner="sum",

dtype=dtypes.string):

SparseFeature 是如何存放这些离散取值的呢？这个跟 bucket\_size 和 lookup\_config 这两个参数相关。在实际定义中，有且只定义其中一个参数。通过使用哪一个参数我们可以把 sparse feature 分成两类，定义 lookup\_config 参数的特征使用一个 in memory 的字典存储 feature 的所有取值，包括后面会讲到的\_SparseColumnKeys，\_SparseColumnVocabulary；定义 bucket\_size 参数的特征使用一个哈希表来存储特征值，特征值通过哈希函数散列到各个桶，包括\_SparseColumnHashed 和\_SparseColumnIntegerized(is\_integerized = True)。

dtype 指定特征值的类型，除了字符串类型 (dtypes.string）之外，spare feature column 还支持 64 位整数类型（dtypes.int64），默认我们认为输入的离散特征是字符串，如果我们定义了 is\_integerized = True，那么我们认为特征是一个整型的 id 型特征，我们可以直接用特征的取值作为特征的 id，而不需要建立一个专门的映射。

combiner 参数对应的是样本维度特征的归一化，如果特征列在单个样本上有多个取值，combiner 参数指定如何对单个样本上特征的多个取值进行归一化。源代码注释中是这样写的：「combiner： A string specifying how to reduce if the sparse column is multivalent」，multivalent 的具体含义在 crossed feature column 的定义中有一个稍微清楚的解释（combiner: A string specifying how to reduce if there are multiple entries in a single row）。combiner 可以指定 3 种归一化方式：sum 对应无归一化，sqrtn 对应 L2 归一化，mean 对应 L1 归一化。通常情况下采用 L2 归一化，模型的准确度相对会更高。

SparseColumn 不能直接作为 DNN 的输入，它只能用于直接构建线性模型的输入：

def \_wide\_embedding\_lookup\_arguments(self, input\_tensor):

return \_LinearEmbeddingLookupArguments( input\_tensor=self.id\_tensor(input\_tensor),

weight\_tensor=self.weight\_tensor(input\_tensor),

vocab\_size=self.length,

initializer=init\_ops.zeros\_initializer(),

combiner=self.combiner)

\_LinearEmbeddingLookupArguments 是一个 namedtuple（A new subclass of tuple with named fields）。input\_tensor 是训练样本集中特征的 id 构成的数组，weight\_tensor 中每个元素对应一个样本中该特征的权重，vocab\_size 是特征取值的个数，intiializer 是特征初始化的函数，默认初始化为 0。

不过看源代码中\_SparseColumn 及其子类并没有使用特征权重：

def weight\_tensor(self, input\_tensor):

"""Returns the weight tensor from the given transformed input\_tensor."""

return None

如果需要为\_SparseColumn 的特征赋予权重，可以使用\_WeightedSparseColumn，构造接口函数为 weighted\_sparse\_column（Create a \_SparseColumn by combing sparse\_id\_column and weight\_column）

class \_WeightedSparseColumn(\_FeatureColumn, collections.namedtuple(

"\_WeightedSparseColumn",["sparse\_id\_column", "weight\_column\_name", "dtype"])):

def \_\_new\_\_(cls, sparse\_id\_column, weight\_column\_name, dtype):

return super(\_WeightedSparseColumn, cls).\_\_new\_\_(cls, sparse\_id\_column, weight\_column\_name, dtype)

\_WeightedSparseColumn 需要 3 个参数：sparse\_id\_column 对应 sparse feature column，是\_SparseColumn 类型的对象，weight\_column\_name 为输入中对应 sparse\_id\_column 的 weight column（input\_fn 返回的 features dict 中需要有一个 weight\_column\_name 的 tensor）dtype 是 weight column 中每个元素的数据类型。这里有几个隐含要求：

（1）dtype 需要能够转换成浮点数类型，否则会抛 TypeError；   
（2）weight\_column\_name 对应的 weight column 可以是一个 SparseTensor，也可以是一个常规的 dense tensor，程序会将 dense tensor 转换成 SparseTensor，但是要求 weight column 最终对应的 SparseTensor 与 sparse\_id\_column 的 SparseTensor 有相同的索引 (indices) 和维度 (dense\_shape)。

\_WeightedSparseColumn 输出特征的 id tensor 和 weight tensor 的函数如下：

def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Inserts a tuple with the id and weight tensors."""

if self.sparse\_id\_column not in columns\_to\_tensors:

self.sparse\_id\_column.insert\_transformed\_feature(columns\_to\_tensors)

weight\_tensor = columns\_to\_tensors[self.weight\_column\_name]

if not isinstance(weight\_tensor, sparse\_tensor\_py.SparseTensor):

# The weight tensor can be a regular Tensor. In such case, sparsify it.

// 我们输入的 weight tensor 可以是一个常规的 Tensor，如通过 tf.Constants 构建的 tensor，

// 这种情况下，会调用 dense\_to\_sparse\_tensor 将 weight\_tensor 转换成 SparseTensor。

weight\_tensor = contrib\_sparse\_ops.dense\_to\_sparse\_tensor(weight\_tensor)

// 最终使用的 weight\_tensor 的数据类型是 float

if not self.dtype.is\_floating:

weight\_tensor = math\_ops.to\_float(weight\_tensor)

// 返回中对应该 WeightedSparseColumn 的一个二元组，二元组的第一个元素是 SparseFeatureColumn 调用

// insert\_transformed\_feature 后的 id\_tensor，第二个元素是 weight tensor。

columns\_to\_tensors[self] = tuple([columns\_to\_tensors[self.sparse\_id\_column],weight\_tensor])

def id\_tensor(self, input\_tensor):

"""Returns the id tensor from the given transformed input\_tensor."""

return input\_tensor[0]

def weight\_tensor(self, input\_tensor):

"""Returns the weight tensor from the given transformed input\_tensor."""

return input\_tensor[1]

## 1.sparse column from keys

这个是最简单的离散特征，类比于枚举类型，一般用于枚举的值不是太多的情况。创建基于 keys 的 sparse 特征的接口是 sparse\_column\_with\_keys(column\_name, keys, default\_value=-1, combiner=None)，对应类是 SparseColumnKeys，构造函数为：

def \_\_new\_\_(cls, column\_name, keys, default\_value=-1, combiner="sum"):

return super(\_SparseColumnKeys, cls).\_\_new\_\_(cls, column\_name, combiner=combiner,

lookup\_config=\_SparseIdLookupConfig(keys=keys, vocab\_size=len(keys),

default\_value=default\_value), dtype=dtypes.string)

keys 为一个字符串列表，定义了所有的枚举值。构造特征输入的 keys 最后存储在 lookup\_config 里面，每个 key 的类型是 string，并且对应 1 个 id，id 是该 key 在输入的 keys 数组中的下标。在模型实际训练中使用的是每个 key 对应的 id。

SparseColumnKeys 输入到模型前需要将枚举值的 key 转换到相应的 id，这个转换工作在函数 insert\_transformed\_feature 中实现：

def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Handles sparse column to id conversion."""

input\_tensor = self.\_get\_input\_sparse\_tensor(columns\_to\_tensors)

""""Returns a lookup table that converts a string tensor into int64 IDs.This operation constructs a lookup table

to convert tensor of strings into int64 IDs. The mapping can be initialized from a string `mapping` 1-D

tensor where each element is a key and corresponding index within the tensor is the

value.

"""

table = lookup.index\_table\_from\_tensor(mapping=tuple(self.lookup\_config.keys),

default\_value=self.lookup\_config.default\_value, dtype=self.dtype, name="lookup")

columns\_to\_tensors[self] = table.lookup(input\_tensor)

## 2.sparse column from vocabulary file

sparse column with keys 一般枚举都能满足，如果枚举的值多了就不合适了，所以提供了一个从文件加载枚举变量的接口：

sparse\_column\_with\_vocabulary\_file((column\_name, vocabulary\_file, num\_oov\_buckets=0, vocab\_size=None,

default\_value=-1, combiner="sum",dtype=dtypes.string)

对应的构造函数为：

def \_\_new\_\_(cls, column\_name, vocabulary\_file, num\_oov\_buckets=0, vocab\_size=None, default\_value=-1,

combiner="sum", dtype=dtypes.string):

那么从文件中读入的特征值是存哪里呢？看看这个构造函数最后返回的类实例：

return super(\_SparseColumnVocabulary, cls).\_\_new\_\_(cls, column\_name,combiner=combiner,

lookup\_config=\_SparseIdLookupConfig(vocabulary\_file=vocabulary\_file,num\_oov\_buckets=num\_oov\_buckets,

vocab\_size=vocab\_size,default\_value=default\_value), dtype=dtype)

如同\_SparseColumnKeys，这个特征也使用了\_SparseIdLookupConfig 来存储特征值，vocabulary\_file 指向定义枚举值的文件，vocabulary\_file 每一行对应一个枚举值，每个枚举值的 id 是该枚举值所在行号（注意，行号是从 0 开始的），vocab\_size 定义枚举值的个数。\_SparseIdLookupConfig 从特征文件中构建一个特征值到 id 的哈希表，我们看看 SparseColumnVocabulary 进行 vocabulary 到 id 的转换时如何使用\_SparseIdLookupConfig 对象。

def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Handles sparse column to id conversion."""

st = self.\_get\_input\_sparse\_tensor(columns\_to\_tensors)

if self.dtype.is\_integer:

// 输入的整数数值型特征转换成字符串形式

sparse\_string\_values = string\_ops.as\_string(st.values)

sparse\_string\_tensor = sparse\_tensor\_py.SparseTensor(st.indices,sparse\_string\_values, st.dense\_shape)

else:

sparse\_string\_tensor = st

"""Returns a lookup table that converts a string tensor into int64 IDs.This operation constructs a lookup table

to convert tensor of strings into int64 IDs. The mapping can be initialized from a vocabulary file specified in

`vocabulary\_file`, where the whole line is the key and the zero-based line number is the ID.

table = lookup.index\_table\_from\_file(vocabulary\_file=self.lookup\_config.vocabulary\_file,

num\_oov\_buckets=self.lookup\_config.num\_oov\_buckets,vocab\_size=self.lookup\_config.vocab\_size,

default\_value=self.lookup\_config.default\_value, name=self.name + "\_lookup")

columns\_to\_tensors[self] = table.lookup(sparse\_string\_tensor)

index\_table\_from\_file 函数从 lookup\_config 的字典文件中构建 table。Table 变量是一个 string 到 int64 的 HashTable，如果定义了 num\_oov\_buckets，table 是 IdTableWithHashBuckets 对象（a string to id wrapper that assigns out-of-vocabulary keys to buckets）。

## 3.sparse column with hash bucket

如果没有 vocab 文件定义枚举特征，我们可以使用 hash bucket 特征，使用该特征的接口是   
sparse\_column\_with\_hash\_bucket(column\_name, hash\_bucket\_size, combiner=None,dtype=dtypes.string)   
对应类\_SparseColumnHashed 的构造函数为：def **new**(cls, column\_name, hash\_bucket\_size, combiner=”sum”, dtype=dtypes.string):

ash\_bucket\_size 定义哈希桶的个数，用于哈希值取模。dtype 支持整数和字符串。实际计算哈希值的时候是将整数转换成对应的字符串表示形式，用字符串计算哈希值然后取模，转换后的特征值是 0 到 hash\_bucket\_size 的一个整数。

def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Handles sparse column to id conversion."""

input\_tensor = self.\_get\_input\_sparse\_tensor(columns\_to\_tensors)

if self.dtype.is\_integer:

// 整数类型的输入转换成字符串类型

sparse\_values = string\_ops.as\_string(input\_tensor.values)

else:

sparse\_values = input\_tensor.values

sparse\_id\_values = string\_ops.string\_to\_hash\_bucket\_fast(sparse\_values, self.bucket\_size, name="lookup")

// Sparse 特征的哈希值作为特征值对应的 id 返回

columns\_to\_tensors[self] = sparse\_tensor\_py.SparseTensor(input\_tensor.indices, sparse\_id\_values,

input\_tensor.dense\_shape)

## 4.integerized sparse column

hash bucket 的 sparse 特征取哈希值的时候是将整数看做字符串处理的，如果我们希望用整数本身的数值作为哈希值，可以使用\_SparseColumnIntegerized，对应的接口是

sparse\_column\_with\_integerized\_feature：

def sparse\_column\_with\_integerized\_feature(column\_name,hash\_bucket\_size,combiner="sum",

dtype=dtypes.int64)

对应的类是\_SparseColumnIntegerized： def \_\_new\_\_(cls, column\_name, bucket\_size, combiner="sum", dtype=dtypes.int64)

特征的转换函数定义：def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Handles sparse column to id conversion."""

input\_tensor = self.\_get\_input\_sparse\_tensor(columns\_to\_tensors)

// 直接对特征值取模，取模后的值作为特征值的 id

sparse\_id\_values = math\_ops.mod(input\_tensor.values, self.bucket\_size, name="mod")

columns\_to\_tensors[self] = sparse\_tensor\_py.SparseTensor( input\_tensor.indices, sparse\_id\_values,

input\_tensor.dense\_shape)

## 5.crossed column

Crossed column 支持 1 个以上的离散型 feature column 进行笛卡尔积，组成高维度的交叉特征。特征之间进行交叉，可以将特征之间的相关性引入模型，增强模型的表达能力。crossed column 仅支持以下 3 种离散特征的交叉组合： \_SparsedColumn, \_BucketizedColumn 和\_CrossedColumn，其接口定义为：

def crossed\_column(columns,hash\_bucket\_size, combiner=」sum」,ckpt\_to\_load\_from=None,

tensor\_name\_in\_ckpt=None, hash\_key=None)

对应类为\_CrossedColumn：

def \_\_new\_\_(cls, columns,hash\_bucket\_size,hash\_key, combiner="sum",ckpt\_to\_load\_from=None,

tensor\_name\_in\_ckpt=None):

columns 对应一个 feature column 的集合，如 tutorial 中的例子：[age\_buckets, education, occupation]；hash\_bucket\_size 参数指定 hash bucket 的桶个数，特征交叉的组合个数越多，hash\_bucket\_size 也应相应增加，从而减小哈希冲突。

交叉特征生成模型输入的逻辑可以分为如下两步：

def insert\_transformed\_feature(self, columns\_to\_tensors):

"""Handles cross transformation."""

def \_collect\_leaf\_level\_columns(cross):

"""Collects base columns contained in the cross."""

leaf\_level\_columns = []

for c in cross.columns:

// 对 CrossedColumn 类型的 feature column 进行递归展开

if isinstance(c, \_CrossedColumn):

leaf\_level\_columns.extend(\_collect\_leaf\_level\_columns(c))

else:

// SparseColumn 和 BucketizedColumn 作为叶子节点

leaf\_level\_columns.append(c)

return leaf\_level\_columns

// 步骤 1： 将 crossed column 中的所有特征进行递归展开，展开后的特征值存放在 feature\_tensors 数组中

feature\_tensors = []

for c in \_collect\_leaf\_level\_columns(self):

if isinstance(c, \_SparseColumn):

feature\_tensors.append(columns\_to\_tensors[c.name])

else:

if c not in columns\_to\_tensors:

c.insert\_transformed\_feature(columns\_to\_tensors)

if isinstance(c, \_BucketizedColumn):

feature\_tensors.append(c.to\_sparse\_tensor(columns\_to\_tensors[c]))

else:

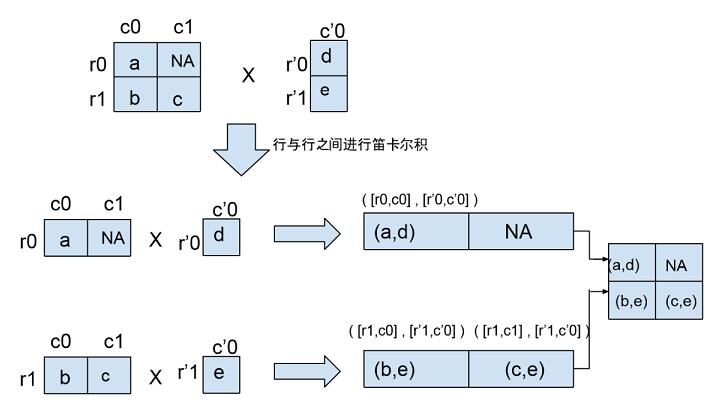
feature\_tensors.append(columns\_to\_tensors[c])

// 步骤 2: 生成 cross feature 的 tensor，sparse\_feature\_cross 通过动态库调用 SparseFeatureCross 函数，函数接

//口可参见 sparse\_feature\_cross\_op.cc

columns\_to\_tensors[self] = sparse\_feature\_cross\_op.sparse\_feature\_cross(feature\_tensors,

hashed\_output=True,num\_buckets=self.hash\_bucket\_size,hash\_key=self.hash\_key, name="cross")



需要指出的一点是：交叉特征是没有权重定义的。

对离散特征进行交叉组合在预测模型中使用比较广泛，但是该类特征的一个局限性是它对训练数据中没有见过的特征组合泛化能力有限，后面我们谈到的 embedding column 则是通过构建离散特征的低维向量表示，强化离散特征的泛化能力。

## 6.real valued column

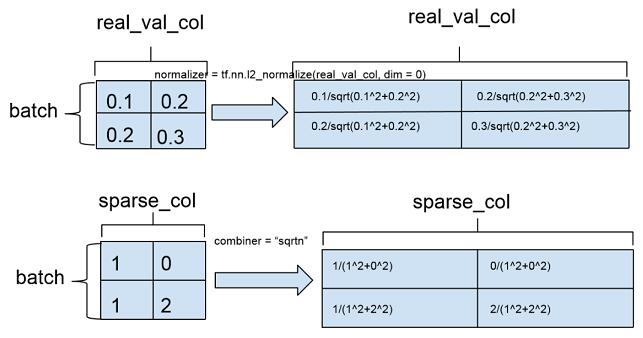
real valued feature column 对应连续型数值特征，接口为

real\_valued\_column(column\_name, dimension=1, default\_value=None, dtype=dtypes.float32,normalizer=None):

对应类为\_RealValuedColumn：

\_RealValuedColumn(column\_name, dimension, default\_value, dtype,normalizer)

dimension 指定 feature column 的维度，默认值为 1，即 1 维浮点数数组。dimension 也可以取大于 1 的整数，对应多维数组。rea valued column 的特征取值类型可以是 float32 或者 int，int 类型在输入到模型之前会转换成 float 类型。normalizer 定义在一批训练样本实例中，特征在列维度的归一化，相当于 column-level normalization。这个同 sparse feature column 的 combiner 不同，combiner 定义的是离散特征在单个样本维度的归一化（example-level normalization），以下示意图举了个例子来说明两者的区别：



normalizer 在 real valued feature column 输入 DNN 时调用：

def insert\_transformed\_feature(self, columns\_to\_tensors):

# Transform the input tensor according to the normalizer function.

// \_normalized\_input\_tensor 调用的是构造 real valued colum 时传入的 normalizer 函数

input\_tensor = self.\_normalized\_input\_tensor(columns\_to\_tensors[self.name])

columns\_to\_tensors[self] = math\_ops.to\_float(input\_tensor)

real valued column 调用\_to\_dnn\_input\_layer 转换为 DNN 的输入。\_to\_dnn\_input\_layer 生成一个二维数组，数组的每一行是一个训练样本的 real valued column 的特征值，该特征值与其他连续型特征拼接后构成 DNN 的输入层。

def \_to\_dnn\_input\_layer(self,input\_tensor,weight\_collections=None,trainable=True,output\_rank=2):

// DNN 的输入必须是 dense tensor，sparse tensor 需要调用 to\_dense\_tensor 转换成 dense tensor

input\_tensor = self.\_to\_dense\_tensor(input\_tensor)

if input\_tensor.dtype != dtypes.float32:

input\_tensor = math\_ops.to\_float(input\_tensor)

// 调用 dense\_inner\_flatten(input\_tensor, output\_rank)。

// output\_rank = 2，输出 [batch\_size, real value column』s input dimension]

return \_reshape\_real\_valued\_tensor(input\_tensor, output\_rank, self.name)

def \_to\_dense\_tensor(self, input\_tensor):

if isinstance(input\_tensor, sparse\_tensor\_py.SparseTensor):

default\_value = (self.default\_value[0] if self.default\_value is not None else 0)

// Sparse tensor 转换成 dense tensor

return sparse\_ops.sparse\_tensor\_to\_dense(input\_tensor, default\_value=default\_value)

// real valued column 直接返回 input tensor

return input\_tensor

## 7.bucketized column

连续型特征通过 bucketization 生成离散特征，连续特征离散化的优点在网上有一些相关讨论，比如餐馆的距离对用户选择的影响，我们通常会将距离划分为若干个区间，如 100 米以内，1 公里以内等，这样小幅度的距离差异不会对我们最终模型的预测造成太大影响，除非距离差异跨域了区间边界。bucketized column 的接口定义为：def bucketized\_column(source\_column, boundaries) 对应类为\_BucketizedColumn，构造函数定义：def **new**(cls, source\_column, boundaries):source\_column 必须是 real\_valued\_column，boundaries 是一个浮点数的列表，而且列表必须是递增序的，比如 boundaries = [0, 100, 200] 定义了以下一组区间：（-INF，0），[0，100），[100，200），[200, INF)。

def insert\_transformed\_feature(self, columns\_to\_tensors):

# Bucketize the source column.

if self.source\_column not in columns\_to\_tensors:

self.source\_column.insert\_transformed\_feature(columns\_to\_tensors)

columns\_to\_tensors[self] = bucketization\_op.bucketize(columns\_to\_tensors[self.source\_column],

boundaries=list(self.boundaries), name="bucketize")

bucketize 函数调用 tensorflow c++ core library 中的 BucketizeOp 类完成 feature 的 bucketization 功能。

## 8.embedding column

sparse feature column 通过 embedding 转换成连续型向量后可以作为 deep model 的输入，前面谈到了 cross column 的一个不足之处是在测试集合的泛化能力，通过 embedding column 将离散特征连续化，根据标注学习特征的向量形式，如同矩阵分解中学习物品的隐含因子向量或者词向量模型中单词的词向量。embedding column 的接口形式是：

def embedding\_column(sparse\_id\_column, dimension, combiner=None, initializer=None,

ckpt\_to\_load\_from=None,tensor\_name\_in\_ckpt=None, max\_norm=None, trainable=True)

对应类为\_EmbeddingColumn：

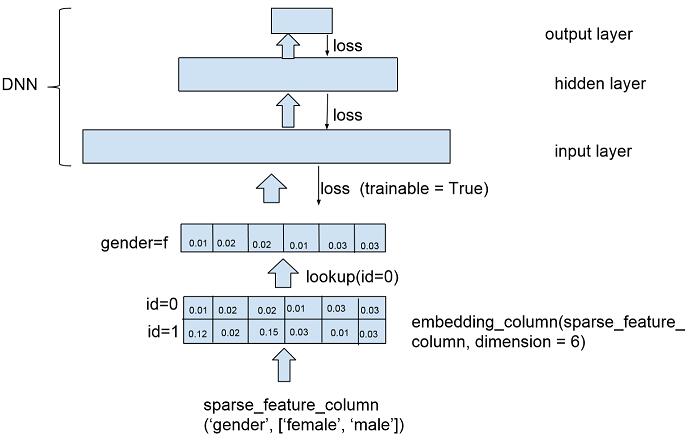
def \_\_new\_\_(cls,sparse\_id\_column,dimension,combiner="mean",initializer=None, ckpt\_to\_load\_from=None,

tensor\_name\_in\_ckpt=None,shared\_embedding\_name=None, shared\_vocab\_size=None,max\_norm=None,

trainable = True):

sparse\_id\_column 是 SparseColumn 对象或者 WeightedSparseColumn 对象，dimension 是 embedding column 的向量维度。SparseColumn 的每个特征取值对应一个整数 id，该整数 id 在 embedding column 中对应一个 dimension 维度的浮点数向量。combiner 参数指定在单个样本上对特征向量归一化的方式，initializer 参数指定特征向量的初始化函数，默认按 truncated normal distribution 初始化 (mean = 0, stddev = 1/ sqrt(length of sparse id column))。max\_norm 限定每个样本特征向量做 L2 归一化后的最大值：embedding\_vector = embedding\_vector \* max\_norm / L2\_norm(embedding\_vector)。

为了进一步理解 embedding column，我们可以画一个简易图：



如上图，以 sparse\_column\_with\_keys(column\_name = 『gender』, keys = [『female』, 『male』]) 为例，假设 female 对应 id = 0, male 对应 id = 1，每个 id 在 embedding feature 中对应 1 个 6 维的浮点数向量。在实际训练数据中，当 gender 特征取值为』female』时，给到 DNN 输入层的将是 id = 0 对应的向量（tf.embedding\_lookup\_sparse）。embedding\_column 设置了一个 trainable 参数，指定是否根据模型训练误差更新特征对应的 embedding。

embedding 特征的变换函数：

def insert\_transformed\_feature(self, columns\_to\_tensors):

if self.sparse\_id\_column not in columns\_to\_tensors:

self.sparse\_id\_column.insert\_transformed\_feature(columns\_to\_tensors)

columns\_to\_tensors[self] = columns\_to\_tensors[self.sparse\_id\_column]

def \_deep\_embedding\_lookup\_arguments(self, input\_tensor):

return \_DeepEmbeddingLookupArguments(

input\_tensor=self.sparse\_id\_column.id\_tensor(input\_tensor),

// sparse\_id\_column 为\_SparseColumn 类型的对象时，weight\_tensor = None

// sparse\_id\_column 为\_WeightedSparseColumn 类型对象时，weight\_tensor = WeihgtedSparseColumn 的

// weight tensor，weight\_tensor 须满足：

// 1）weight\_tensor.indices = input\_tensor.indices

// 2）weight\_tensor.shape = input\_tensor.shape

weight\_tensor=self.sparse\_id\_column.weight\_tensor(input\_tensor),

// sparse feature column 的元素个数

vocab\_size=self.length,

// embedding 的维度

dimension=self.dimension,

// embedding 的初始化函数

initializer=self.initializer,

// embedding 的行归一化方法

combiner=self.combiner,

shared\_embedding\_name=self.shared\_embedding\_name,

hash\_key=None,

max\_norm=self.max\_norm,

trainable=self.trainable)

从\_DeepEmbeddingLookupArguments 产生 sparse feature 的 embedding 的逻辑在函数\_embeddings\_from\_arguments 实现:

def \_embeddings\_from\_arguments(column, args, weight\_collections,trainable, output\_rank=2):

// column 对应 embedding feature column 的 name，args 是 feature column 对应的

// \_DeepEmbeddingLookupArguments 对象，weight\_collections 存储 embedding 的权重，

// output\_rank 指定输出 embedding 的 tensor 的 rank。

input\_tensor = layers.\_inner\_flatten(args.input\_tensor, output\_rank)

weight\_tensor = layers.\_inner\_flatten(args.weight\_tensor, output\_rank)

// 考虑默认情况下构建 embedding: args.hash\_key is None, args.shared\_embedding\_name is None

// 获取或创建 embedding 的 model variable

// embeddings 是 [number of sparse feature id, embedding dimension] 的浮点数二维数组

// 每行对应一个 sparse feature id 的 embedding

embeddings = contrib\_variables.model\_variable( name='weights'，shape=[args.vocab\_size,

args.dimension], dtype=dtypes.float32,initializer=args.initializer,

// If trainable, embedding vector 作为一个 model variable 添加到 GraphKeys.TRAINABLE\_VARIABLES

trainable=(trainable and args.trainable),

collections=weight\_collections // weight\_collections 存储每个 feature id 的 weight

)

// 获取每个 sparse feature id 的 embedding

return embedding\_ops.safe\_embedding\_lookup\_sparse(embeddings, input\_tensor,

sparse\_weights=weight\_tensor, combiner=args.combiner, name=column.name + 'weights',

max\_norm=args.max\_norm)

safe\_embedding\_lookup\_sparse 调用 tf.embedding\_lookup\_sparse 获取每个 sparse feature id 的 embedding。   
tf.embedding\_lookup\_sparse 首先调用 tf.embedding\_lookup 获取 sparse feature id 的 embedding vector:

// sp\_ids 是 input\_tensor 的 id tensor

ids = sp\_ids.values

embeddings = embedding\_lookup (

// params 对应 embeddings 矩阵，每个元素是 embedding\_dimension 的 float tensor，可以将 params 看

// 做一个 embedding tensor 的 partitions，partition 的策略由 partition\_strategy 指定

params,

// ids 对应 input\_tensor 的 values 数组

ids,

// id 分配到 params 的分配策略，有 mod 和 div 两种，默认 mod，具体定义可参见 tf.embedding\_lookup 的说明

partition\_strategy=partition\_strategy,

// 限制 embedding 的最大 L2-Norm

max\_norm=max\_norm

)

如果 sparse\_weights 不是 None，embedding 的值乘以 weights，   
weights = sparse\_weights.values   
embeddings \*= weights

根据 combiner，对 embedding 进行归一化

segment\_id = sp\_ids.indices[;0]

if combiner == "sum":

// No normalization

embeddings = math\_ops.segment\_sum(embeddings, segment\_ids, name=name)

elif combiner == "mean":

// L1 normlization: embeddings = SUM(embeddings \* weight) / SUM(weight)

embeddings = math\_ops.segment\_sum(embeddings, segment\_ids)

weight\_sum = math\_ops.segment\_sum(weights, segment\_ids)

embeddings = math\_ops.div(embeddings, weight\_sum, name=name)

elif combiner == "sqrtn":

// L2 normalization: embeddings = SUM(embeddings \* weight^2) / SQRT(SUM(weight^2))

embeddings = math\_ops.segment\_sum(embeddings, segment\_ids)

weights\_squared = math\_ops.pow(weights, 2)

weight\_sum = math\_ops.segment\_sum(weights\_squared, segment\_ids)

weight\_sum\_sqrt = math\_ops.sqrt(weight\_sum)

embeddings = math\_ops.div(embeddings, weight\_sum\_sqrt, name=name)

## 9.其他 feature columns

除了以上列举的几个 feature column，TensorFlow 还支持 one hot column，shared embedding column 和 scattered embedding column。one hot column 对 sparse feature column 进行 one-hot 编码，如果离散特征的取值较少，可以用 one hot feature column 进行编码用于 DNN 的训练。不同于 embedding column，one hot feature column 不支持通过模型训练来更新其特征的 embedding。shared embedding column 和 scattered embedding column 由于篇幅原因就不多谈了。

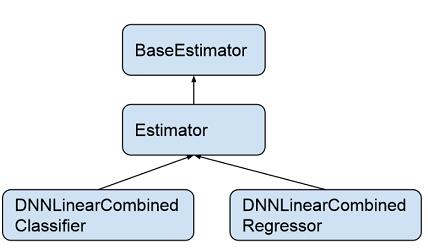
# 四．训练实现

**前面讲了模型输入的特征，下面谈谈模型本身。**关于 wide and deep 模型官方教程中有一段描述：The wide models and deep models are combined by summing up their final output log odds as the prediction, then feeding the prediction to a logistic loss function。从这里大概看出，线性模型与 DNN 模型进行结合的方式是对两者的预测结果进行相加，然后把相加过后的值拿去计算分类的 loss。下面我们在源代码级别把上面这段描述展开，详细分析 wide 端模型和 deep 端模型是如何实现结合的。

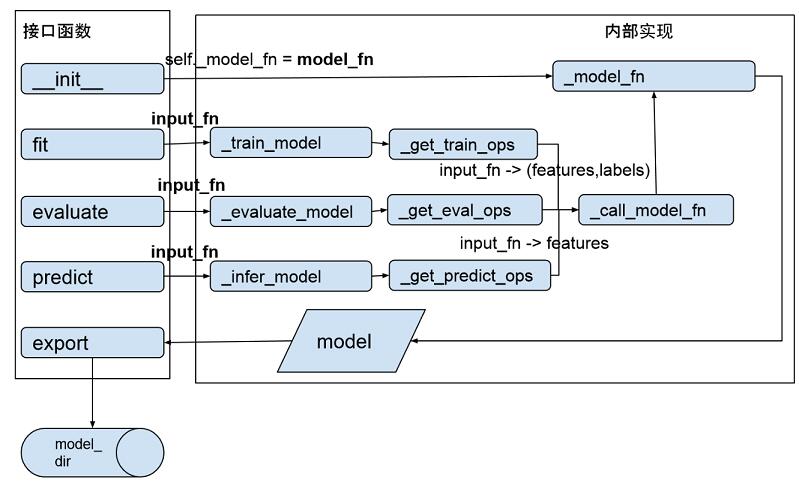
由于我们运用的只是分类模型，所以就不对回归模型进行分析。

## 1.内部函数的调用过程

DNNLinearCombinedClassifier 类继承于类 Estimator，Estimator 类继承于类 BaseEstimator。BaseEstimator 是一个抽象类，定义了通用的模型训练以及评测的函数接口 (train\_model, evaluate\_model, infer\_model)，Estimator 类中用一个统一函数 call\_model\_fn 来实现 train\_model, evaluate\_model, infer\_model。

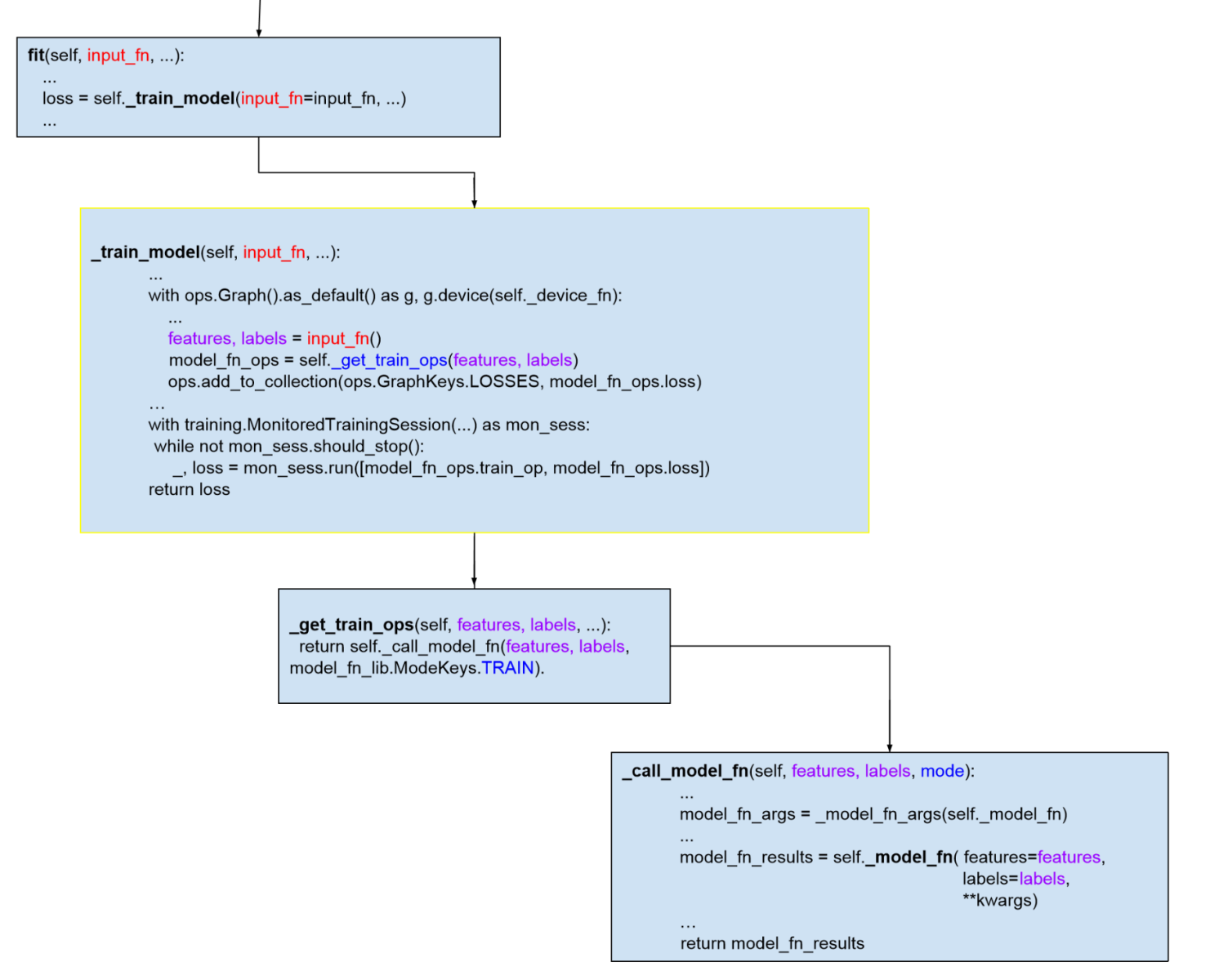


为了更好了解整个过程，我们看看内部函数的调用过程（代码可以参见 estimator/estimator.py）：



模型训练通过调用 BaseEstimator 的 fit() 接口开始，其调用栈是：fit -> \_train\_model -> \_get\_train\_ops ->\_call\_model\_fn(ModelKeys.TRAIN) -> \_model\_fn，最终\_model\_fn() 产生模型并通过 export 函数将模型输出到 model\_dir 对应目录中。

我们把训练模型的调用过程在代码级别展开，标出关键的几个函数和数据结构，省略不关键的代码，希望能让读者看到训练模型的大致过程：



评测（evaluate）和预测（predict）的过程与训练（train）大致相同，读者可以通过源代码文件找到对应函数了解。可以看出，整个函数调用栈中最关键的 2 个函数是： input\_fn 和 model\_fn。input\_fn 从输入数据中生成 features 和 labels，features 是一个 Tensor 或者是一个从特征名到 Tensor 的字典，如果 features 是一个 Tensor，程序会给这个 Tensor 一个空字符串的键值，转换成特征名到 Tensor 的字典。labels 是样本的 label 构成的 tensor。input\_fn 由应用程序调用者提供实现，返回（features, labels）二元组，要求 tf.get\_shape(features)[0] == tf.get\_shape(labels)[0]，也就是两个 tensor 的行数目得保持一致。model\_fn 定义训练和评测模型的具体逻辑，如模型训练产生的误差 (model\_fn\_ops.loss) 以及训练算子（model\_fn\_ops.train\_op）通过封装在 EstmiatorSpec 的对象中由 training 的 Session 进行调用。每个具体模型需要实现的是自定义的 model\_fn。

DNNLinearCombinedClassifier 是如何实现自己的 model\_fn 的呢？本文开头我们给出了它的初始化函数原型，进入初始化函数的实现中我们定位到代码行 model\_fn=\_dnn\_linear\_combined\_model\_fn。

这个就是 DNNLinearCombinedClassifier 的 model\_fn。这个函数的定义如下：

def \_dnn\_linear\_combined\_model\_fn(features, labels, mode, params, config= None)

features 和 labels 大家都已经知道，mode 指定 model\_fn 的操作模式，目前支持 3 个值：训练模型 (model\_fn.ModeKeys.TRAIN)，对模型进行评测 (model\_fn.ModeKeys.EVAL)，根据输入特征进行预测 (model\_fn.ModeKeys.PREDICT)，mode 的定义可参见文件 estimator/model\_fn.py。params 和 config 参数分别定义模型训练的参数以及模型运行的配置。

## ****2.模型训练的步骤****

### ****2.1从 params 获取参数，定义模型优化函数****

// 将输入的特征统一为字符串到 tensor 的 dict

features = \_get\_feature\_dict(features)

// 定义线性模型的优化函数

linear\_optimizer = params.get("linear\_optimizer") or "Ftrl"

linear\_optimizer = \_get\_optimizer(linear\_optimizer)

// 定义 DNN 模型的优化函数

dnn\_optimizer = params.get("dnn\_optimizer") or "Adagrad"

dnn\_optimizer = \_get\_optimizer(dnn\_optimizer)

### ****2.2构建 DNN 模型****

首先从 dnn\_feature\_columns 构造 DNN 模型输入层：

dnn\_feature\_columns = params.get("dnn\_feature\_columns")

// dnn\_feature\_columns 中包括 DNN 模型需要的所有连续特征和 embedding 特征

net = layers.input\_from\_feature\_columns(columns\_to\_tensors=features,

feature\_columns=dnn\_feature\_columns,

weight\_collections=[dnn\_parent\_scope],

scope=dnn\_input\_scope)

这里的 features 是 feature column key 到 tensor 的 dict，在本文前面我们对每个特征都谈了它的 insert\_transformed\_feature(columns\_to\_tensors)，在构建 DNN 模型输入层之前会调用每个 FeatureColumn 的 insert\_transformed\_feature 函数生成一个浮点数类型的 tensor 作为 DNN 输入层的一部分。

构建完输入层后，然后至底向上进行隐藏层构建，每层的隐藏单元的个数在参数

dnn\_hidden\_units 中定义：

dnn\_hidden\_units = params.get("dnn\_hidden\_units")

for layer\_id, num\_hidden\_units in enumerate(dnn\_hidden\_units): // 从输入层或者是下面一层隐藏层构建新的隐藏层 // dnn\_activation\_fn 定义隐藏层的激活函数，默认使用 RELU

dnn\_activation\_fn = params.get("dnn\_activation\_fn") or nn.relu。

net = layers.fully\_connected(net，num\_hidden\_units，activation\_fn=dnn\_activation\_fn，

variables\_collections=[dnn\_parent\_scope], scope=dnn\_hidden\_layer\_scope)

// 模型训练中对隐藏层进行 drop-out // dnn\_dropout 是舍弃隐藏单元输出的概率

dnn\_dropout = params.get("dnn\_dropout")

if dnn\_dropout is not None and mode == model\_fn.ModeKeys.TRAIN:

net = layers.dropout(net, keep\_prob=(1.0 - dnn\_dropout))

从最后一层隐藏层到输出层建立一个全连接就完成 DNN 模型的构建了, dnn\_logits 是模型输出，它是一个分类数个数的 Tensor，Tensor 的每个元素对应 1 个分类的线性激活值（linear activation: w \* net + bias）：

with variable\_scope.variable\_scope( "logits", values=(net,)) as dnn\_logits\_scope:

dnn\_logits = layers.fully\_connected( net,

head.logits\_dimension, // 分类个数

activation\_fn=None, // 输出层不做非线性变换

variables\_collections=[dnn\_parent\_scope],

scope=dnn\_logits\_scope)

从 dnn feature columns 中构建 DNN 输入层的工作主要由函数 layers.\_input\_from\_feature\_columns 完成，每个训练样本上的 dnn feature column 的值拼接成一个 dense tensor 作为 DNN 的输入。

该函数的定义如下：

def \_input\_from\_feature\_columns( columns\_to\_tensors, feature\_columns, weight\_collections,

trainable, scope, output\_rank, default\_name)

transformer = \_Transformer(columns\_to\_tensors)

for column in sorted(set(feature\_columns), key=lambda x: x.key):

// 调用 feature column 的特征变换函数 insert\_transformed\_feature()

// 返回 feature column 变换后的 tensor： columns\_to\_tensors[column]。

transformed\_tensor = transformer.transform(column)

try:

// 构建\_EmbeddingColumn 的 embedding

arguments = column.\_deep\_embedding\_lookup\_arguments(transformed\_tensor)

output\_tensors.append(\_embeddings\_from\_arguments(column, arguments, weight\_collections,

trainable, output\_rank=output\_rank))

except NotImplementedError as ee:

// 构建\_RealValuedColumn 的 tensor

try:

output\_tensors.append(column.\_to\_dnn\_input\_layer(transformed\_tensor,weight\_collections,

trainable,output\_rank=output\_rank))

except ValueError as e:

// output\_tensors 数组每个元素对应一个 feature column 在该批次训练数据中每个训练实例上生成的 tensor，

// 根据 output\_rank - 1 对 output\_tensor 中的每个 tensor 在指定维度上进行拼接

return array\_ops.concat(output\_tensors, output\_rank - 1)

举一个例子说明下拼接的结果吧，假设该批次有 2 个训练实例，real\_val\_col 对应一个 real\_valued\_column，deep\_emb\_col 对应一个 embedding\_column，real\_val\_col 和 deep\_emb\_col 的第一个元素分别对应第一个训练实例上的 feature column 的 tensor，第二个元素对应第二个训练实例上的 feature column 的 tensor，可以参看以下演示代码的执行结果：

real\_val\_col = [[0.1,0.2], [0.3,0.4]]

deep\_emb\_col = [[0.01,0.02,0.03], [0.04,0.05,0.06]]

output\_tensors = []

output\_tensors.append(real\_val\_col)

output\_tensors.append(deep\_emb\_col)

output\_rank = 2

dnn\_input = array\_ops.concat(output\_tensors，output\_rank - 1)

tf.InteractiveSession().run(c)

array([

[ 0.1 , 0.2, 0.01, 0.02, 0.03],

[ 0.30000001, 0.40000001, 0.04, 0.05,0.06]], dtype=float32)

可以看到最终拼接的结果是 [

[real\_val\_col[0], deep\_emb\_col[0]]

[real\_val\_col[1], deep\_emb\_col[1]]

### ****2.3构建线性模型****

线性模型的公式是：y = W \* x + b，x 从 linear\_feature\_columns 中构建，W 是各个特征的权重，b 是模型的偏置。每个分类的预测值存储在 linear\_logits 变量中。

linear\_logits, \_, \_ = layers.weighted\_sum\_from\_feature\_columns(columns\_to\_tensors=features,

feature\_columns=linear\_feature\_columns,

num\_outputs=head.logits\_dimension, // 分类数

weight\_collections=[linear\_parent\_scope],

scope=scope)

weighted\_sum\_feature\_columns() 函数执行过程同构建 DNN 输入层的\_input\_from\_feature\_columns 函数执行过程比较相似，以下是它的源代码：

// 每个特征与权重矩阵相乘得到每个分类上该特征的预测分数，将每个分类上特征的预测分数累加后加上偏置得到

// 对每个分类的最终预测值，如图 10for column in sorted(set(feature\_columns), key=lambda x: x.key):

transformed\_tensor = transformer.transform(column)

try: // 线性模型使用\_wide\_embedding\_lookup\_arguments 返回的 LinearEmbeddingArguments 构建权重矩阵 // SparsColumn, WeightedSparseColumn 或 Crossed Column 定义了 wide\_embedding\_lookup\_arguments

embedding\_lookup\_arguments = column.\_wide\_embedding\_lookup\_arguments( transformed\_tensor)

// create\_embedding\_lookup() 的实现同 embedding\_lookup\_arguments() 比较类似，variable 中返回权重矩 // 阵 W，predictions 返回 W \* X

variable, predictions = \_create\_embedding\_lookup(column, columns\_to\_tensors,

embedding\_lookup\_arguments, num\_outputs,

trainable, weight\_collections)

except NotImplementedError: // 其他特征直接作为线性模型的输入 // 如果输入是 Sparse tensor，需要转换为 dense tensor。

tensor = column.\_to\_dense\_tensor(transformed\_tensor)

// 展开成二维数组 [batch\_size, dimension of feature column]

tensor = \_maybe\_reshape\_input\_tensor( tensor, column.name, output\_rank=2)

// variable 定义线性模型的权重 W // W 是二维矩阵，行是特征的取值个数，列是分类个数 // W 初始化为零矩阵 // 设置 trainable=True, W 加入 GraphKeys.TRAINABLE\_VARIABLES

variable = [ contrib\_variables.model\_variable(name='weight',

shape=[tensor.get\_shape()[1], num\_outputs],

initializer=init\_ops.zeros\_initializer(),

trainable=trainable,

collections=weight\_collections)] // y』 = W \* x

predictions = math\_ops.matmul(tensor, variable[0], name='matmul')

output\_tensors.append(array\_ops.reshape(predictions, shape=(-1, num\_outputs)))

column\_to\_variable[column] = variable

//在每个分类上，将所有 feature column 对该分类的预测值相加

predictions\_no\_bias = math\_ops.add\_n(output\_tensors)

// bias 是模型的偏置项，是一个一维向量，向量大小是输出分类的个数，初始化为 0 // 设置 Trainable=True, bias 加入 GraphKeys.TRAINABLE\_VARIABLES

bias = contrib\_variables.model\_variable('bias\_weight',shape=[num\_outputs],

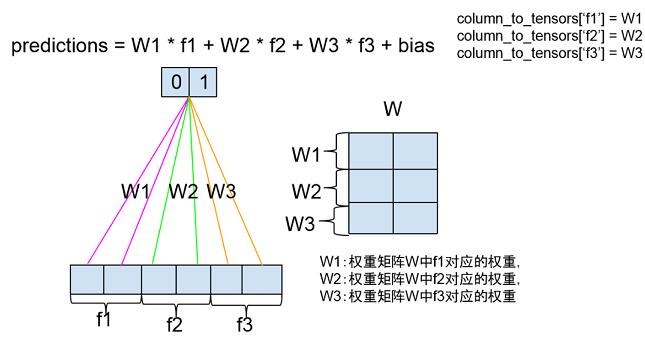
initializer=init\_ops.zeros\_initializer(),trainable=trainable,

collections=\_add\_variable\_collection(weight\_collections))

// predictions: W\*x+b

predictions = nn\_ops.bias\_add(predictions\_no\_bias, bias)

return predictions, column\_to\_variable, bias



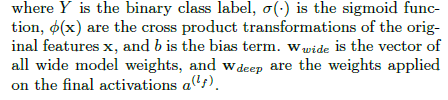
### ****2.4combine****

接下来将两个模型结合成一个模型：

if dnn\_logits is not None and linear\_logits is not None:

logits = dnn\_logits + linear\_logits

在原论文中给了 logistic regression 二分类情况下融合模型的预测公式：

σ函数将模型预测结果进行变换生成各个分类的概率。

### 2.5参数更新

模型产生的误差分别反馈给 DNN 和线性模型进行参数更新：

def \_make\_training\_op(training\_loss):

…

// training\_loss 反馈到 DNN 模型

// DNNLinearCombinedClassifier 中定义的 dnn\_optimizer 用在这里

if dnn\_logits is not None:

train\_ops.append( optimizers.optimize\_loss( loss=training\_loss,learning\_rate=\_DNN\_LEARNING\_RATE,

optimizer=dnn\_optimizer, …)

// training\_loss 反馈到线性模型，DNNLinearCombinedClassifier 中定义的 linear\_optimizer 用在这

if linear\_logits is not None:

train\_ops.append( optimizers.optimize\_loss( loss=training\_loss,

learning\_rate=\_linear\_learning\_rate(len(linear\_feature\_columns)),

optimizer=linear\_optimizer, ... )

return head.create\_model\_fn\_ops(features=features,mode=mode,labels=labels,

train\_op\_fn=\_make\_training\_op, logits=logits)

head.create\_model\_fn\_ops 完成模型算子的定义，包括损失函数计算，每个分类概率的计算，模型评测指标的计算等。DNNLinearCombinedClassifier 在初始化时根据分类个数创建 head 成员变量，如果是二分类，创建\_BinaryLogisticHead 对象，如果是多分类，创建\_MultiClassHead 对象。head 变量的类型定义可参考源代码文件：head.py。

head = head\_lib.multi\_class\_head( n\_classes=n\_classes,weight\_column\_name=weight\_column\_name,

enable\_centered\_bias=enable\_centered\_bias)

我们看看二分类的\_BinaryLogisticHead 类如何生成 model\_fn：

def create\_model\_fn\_ops(self, features,mode,labels=None, train\_op\_fn=None,logits=None,logits\_input=None,

scope=None):

with variable\_scope.variable\_scope(scope, default\_name=self.head\_name or "binary\_logistic\_head",

values=(tuple(six.itervalues(features)) + (labels, logits, logits\_input))):

// 生成 label tensor，维度是 1 个 batch 中训练样本的个数，每个元素是一个样本的标注

labels = self.\_transform\_labels(mode=mode, labels=labels)

// 生成 logits tensor，维度与 label tensor 保持一致，每个元素是一个训练样本的模型输出值

logits = \_logits(logits\_input, logits, self.logits\_dimension)

// 构建模型训练算子

return \_create\_model\_fn\_ops(features=features, mode=mode,

loss\_fn=self.\_loss\_fn, // 损失函数，默认使用交叉熵损失 (\_log\_loss\_with\_two\_classes)

logits\_to\_predictions\_fn = self.\_logits\_to\_predictions, // 计算每个分类的概率，对应上面提到的σ函数

metrics\_fn=self.\_metrics, // 模型指标计算，包括 AUC，accuracy 等，见 MetricKey 的定义

...

labels=labels,

train\_op\_fn=train\_op\_fn, // 根据训练误差进行模型参数的更新函数

logits=logits, // 模型的预测值

...)

二分类默认使用 sigmoid cross entropy 计算损失函数，cross entropy 乘以训练样本的权重得到训练样本的损失函数值。在批次训练中，一批训练样本的损失函数值定义为：weighted\_loss = ( sum { weight[i] \* loss[i]} ) / N，N 是该批训练样本的个数。

def \_log\_loss\_with\_two\_classes(labels, logits, weights=None):

with ops.name\_scope(None, "log\_loss\_with\_two\_classes", (logits, labels)) as name:

logits = ops.convert\_to\_tensor(logits)

labels = math\_ops.to\_float(labels) // label 转换为 float 类型的数组

...

loss = nn.sigmoid\_cross\_entropy\_with\_logits(labels=labels, logits=logits, name=name)

return \_compute\_weighted\_loss(loss, weights)

\_logits\_to\_predictions 函数根据 PredictionKey 计算每个分类的概率或者输出最大概率的分类，logits 是模型未经变换的输出：

def \_logits\_to\_predictions(self, logits):

with ops.name\_scope(None, "predictions", (logits,)):

two\_class\_logits = \_one\_class\_to\_two\_class\_logits(logits)

return {

// LOGITS：直接输出模型对每个分类的预测值

prediction\_key.PredictionKey.LOGITS: logits,

// LOGISITC: simgoid 变换

prediction\_key.PredictionKey.LOGISTIC:

math\_ops.sigmoid(logits, name=prediction\_key.PredictionKey.LOGISTIC),

// SOFTMAX: softmax 变换

prediction\_key.PredictionKey.PROBABILITIES:

nn.softmax( two\_class\_logits, name=prediction\_key.PredictionKey.PROBABILITIES),

// CLASSES: 预测值最大的类别

prediction\_key.PredictionKey.CLASSES:

math\_ops.argmax( two\_class\_logits, 1, name=prediction\_key.PredictionKey.CLASSES)

}

最终构建模型训练算子

def \_create\_model\_fn\_ops(features, mode, loss\_fn, logits\_to\_predictions\_fn, metrics\_fn,

create\_output\_alternatives\_fn, labels=None, train\_op\_fn=None,

logits=None, logits\_dimension=None, head\_name=None,

weight\_column\_name=None,enable\_centered\_bias=False):

// 源代码中对 enable\_centered\_bias 的解释：

// enable\_centered\_bias: A bool. If True, estimator will learn a centered bias variable for each class. Rest of

// the model structure learns the residual after centered bias.

// 融合模型中，centered bias 相当于在每个分类的输出中加上 bias variable:

// logits = (dnn\_logits + linear\_logits) + centered\_bias，

// 分别为 (dnn\_logits+linear\_logits) 以及 centered\_bias 计算 loss 生成 training op

if enable\_centered\_bias:

centered\_bias = \_centered\_bias(logits\_dimension, head\_name)

{

// centered\_bias 的第一个维度是分类个数，初始化为 0，trainable 设置为 True

centered\_bias = variable\_scope.variable(name="centered\_bias\_weight",

initial\_value=array\_ops.zeros(shape=(logits\_dimension,)), trainable=True)

return centered\_bias

}

logits = nn.bias\_add(logits, centered\_bias)

predictions = logits\_to\_predictions\_fn(logits)

// 计算模型的损失

if (mode != model\_fn.ModeKeys.INFER) and (labels is not None):

// input\_fn 返回的 features dictionary 中 weight\_column\_name 列对应的是 weight tensor

weight\_tensor = \_weight\_tensor(features, weight\_column\_name)

// logits 对应 dnn\_logits + linear\_logits + centered\_bias

loss, weighted\_average\_loss = loss\_fn(labels, logits, weight\_tensor)

//根据 loss 进行参数更新，计算评测指标

if mode == model\_fn.ModeKeys.TRAIN:

batch\_size = array\_ops.shape(logits)[0]

// 生成 training op

train\_op = \_train\_op(loss, labels, train\_op\_fn, centered\_bias, batch\_size, loss\_fn, weight\_tensor)

// 生成 metric op

eval\_metric\_ops = metrics\_fn(weighted\_average\_loss, predictions, labels, weight\_tensor)

// 返回模型算子

return model\_fn.ModelFnOps( mode=mode, predictions=predictions, // 分类预测值

loss=loss, // 模型的误差损失

train\_op=train\_op, // 模型训练算子，定义模型参数的更新函数，包括 DNN 和线

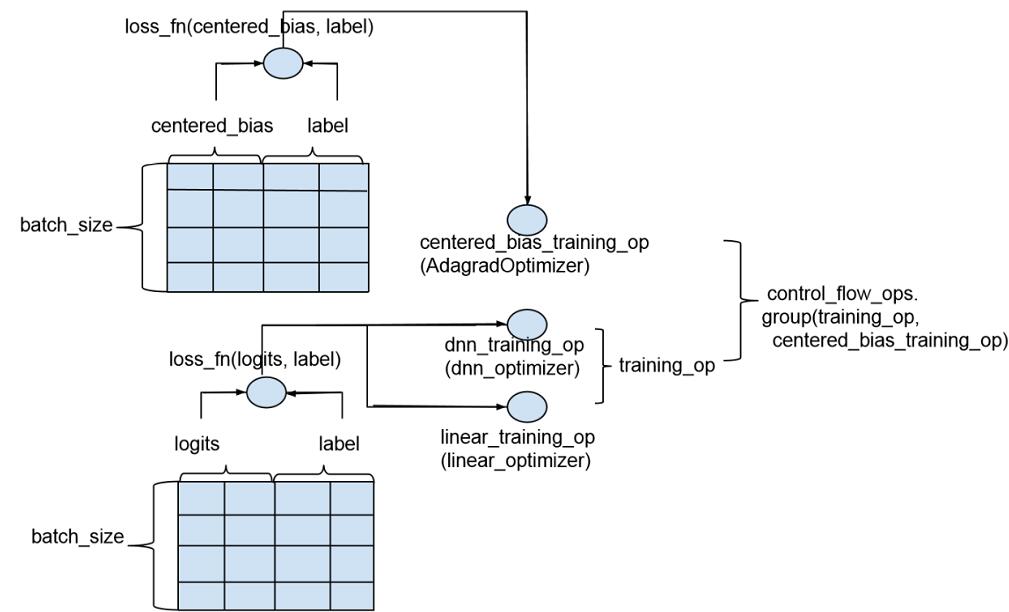
// 性模型的参数以及 centered\_bias 参数

eval\_metric\_ops=eval\_metric\_ops, // 模型指标评测算子

output\_alternatives=create\_output\_alternatives\_fn(predictions))

}

\_train\_op 函数生成 training 的算子，如果 centered\_bias 没有定义 (enable\_centered\_bias=False)，返回 train\_op\_fn 创建的 training op； 如果定义了 centered\_bias，为 centered\_bias 创建 training op，并与 train\_op\_fn 创建的 training\_op 通过 control\_flow\_ops.group 组成组合算子返回。centered\_bias 的 training op 采用 AdagradOptimizer 作为优化函数。下图描述了 enable\_centered\_bias=True 的情况下 training op 的创建过程：



def \_train\_op(loss, labels, train\_op\_fn, centered\_bias, batch\_size, loss\_fn, weights)

{

if centered\_bias is not None:

centered\_bias\_step = \_centered\_bias\_step(centered\_bias=centered\_bias, batch\_size=batch\_size,

labels=labels, loss\_fn=loss\_fn, weights=weights)

else:

centered\_bias\_step = None

with ops.name\_scope(None, "train\_op", (loss, labels)):

train\_op = train\_op\_fn(loss)

if centered\_bias\_step is not None:

train\_op = control\_flow\_ops.group(train\_op, centered\_bias\_step)

return train\_op

}

def \_centered\_bias\_step(centered\_bias, batch\_size, labels, loss\_fn, weights):

"""Creates and returns training op for centered bias."""

with ops.name\_scope(None, "centered\_bias\_step", (labels,)) as name:

logits\_dimension = array\_ops.shape(centered\_bias)[0]

// logits 是 [batch\_size，num of output classes] 的二维数组，数组中每个元素的值是 centered\_bias

logits = array\_ops.reshape(array\_ops.tile(centered\_bias, (batch\_size,)),(batch\_size, logits\_dimension))

with ops.name\_scope(None, "centered\_bias", (labels, logits)):

// 对 centered bias variable 计算 loss，centered\_bias\_loss 是一个 batch 中 loss 的平均值

centered\_bias\_loss = math\_ops.reduce\_mean(loss\_fn(labels, logits, weights), name="training\_loss")

# Learn central bias by an optimizer. 0.1 is a convervative learning rate for a single variable.

// centered\_bias\_loss 是 centered\_bias 同 label 比较后产生的 loss，同 training 模型产生的 loss 是独立的

return training.AdagradOptimizer(0.1).minimize(centered\_bias\_loss, var\_list=(centered\_bias,),

name=name)

}

模型的评价指标的计算在 metrics\_fn 函数中完成，在 BinaryLogisticHead 中 metrics\_fn 对应\_metrics 函数：

// eval\_loss: \_compute\_weighted\_loss() 返回一个 batch 中 per example loss 的平均值

// prediction: \_logits\_to\_prediction() 返回一个 batch 中 per example 的 prediction

// labels: 样本标注，weights: 样本权重 def \_metrics(self, eval\_loss, predictions, labels, weights):

"""Returns a dict of metrics keyed by name."""

with ops.name\_scope("metrics", values=([eval\_loss, labels, weights] + list(six.itervalues(predictions)))):

classes = predictions[prediction\_key.PredictionKey.CLASSES]

logistic = predictions[prediction\_key.PredictionKey.LOGISTIC]

metrics = {\_summary\_key(self.head\_name, mkey.LOSS): metrics\_lib.streaming\_mean(eval\_loss)}

metrics[\_summary\_key(self.head\_name, mkey.ACCURACY)] =

(metrics\_lib.streaming\_accuracy(classes, labels, weights))

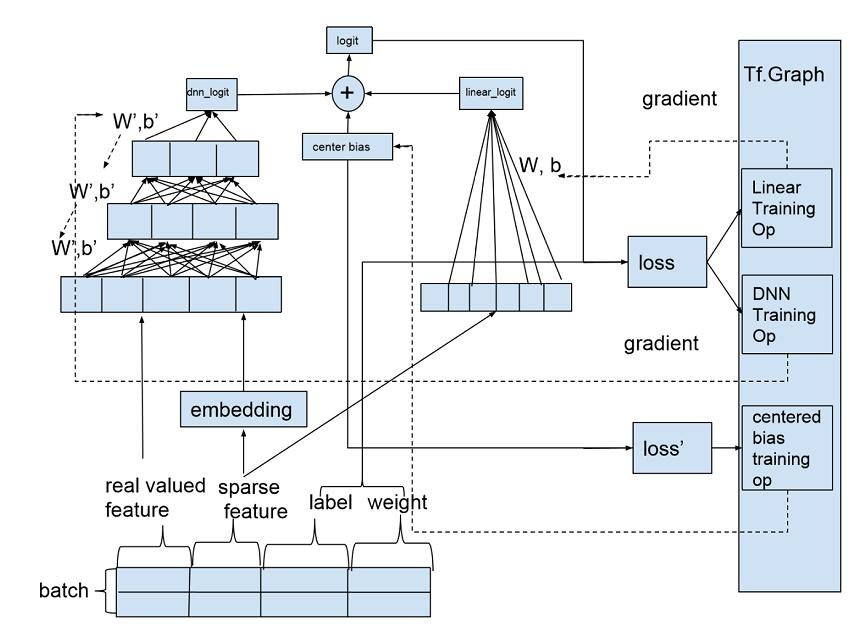
metrics[\_summary\_key(self.head\_name, mkey.AUC)] = ( \_streaming\_auc(logistic, labels, weights))

metrics[\_summary\_key(self.head\_name, mkey.AUC\_PR)] = ( \_streaming\_auc(logistic, labels, weights,

curve="PR"))

return metrics

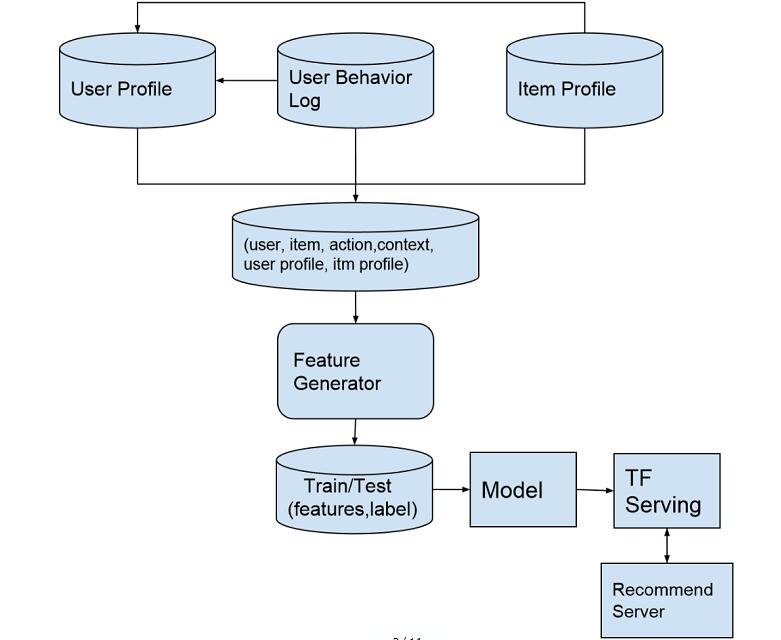
到此 wide and deep 分类模型的具体细节已讲述完毕。下图将 wide and deep 模型的训练过程进行简单总结，希望对大家理解整个训练过程有所帮助。



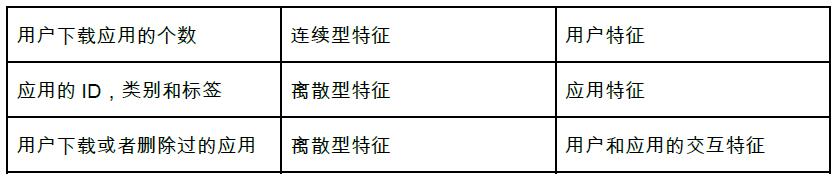
# 五．Wide And Deep 模型应用

接下来考虑将模型应用到产品具体场景。我们通过 DNNLinearCombinedClassifier 构建二分类模型以最大化用户下载应用的概率。

训练和测试数据来自于半年中用户的行为日志，包括下载，使用和删除等操作行为，每条操作中记录用户 ID，应用 ID，操作行为标识，操作时间等参数，我们在用户行为基础上基于应用的标签和类别属性构建用户画像，用户的行为日志最终分别通过用户 ID 和应用 ID 关联用户画像和应用属性生成训练的输入数据：



准备好输入数据后，下一步定义每条数据的标记，特征以及权重。用户下载了应用标记为 1（action = click），用户浏览了应用但是没有下载行为，则标记为 0（action = skip）。标记为 0 的训练实例的权重设定为 1.0，标记为 1 的实例权重通过计算用户下载该应用后的使用时间以及用户后来是否删除该应用来获取：1.0 + min(2, log(使用时间) \* exp(-用户是否删除了该应用))。特征部分主要使用三种类型的特征：用户特征，应用特征以及用户和应用的交互特征。如论文中建议，我们将 embedding 特征和连续型特征作为 DNN 的输入，将交叉特征作为线性模型的输入。下表中列举了我们使用的部分特征：



每条训练实例中的应用与用户历史操作过的应用集合进行交叉构建交叉特征，包括：   
（训练实例中用户看到的应用）交叉（用户之前的历史下载应用）   
（训练实例中用户看到的应用）交叉（用户之前的历史删除应用）

在构建交叉特征时，原论文中提到的是 user impressioned app 同 user installed app 进行交叉，实际中我们需要的是将用户下载历史中的多个 app 与 impressioned app 一一进行交叉，通过一个 crossed column 是无法完成的，我们采取的方式是将用户下载历史中的应用按照下载时间排序，取最近下载的 20 个应用，标记为 1 到 20，然后用每个应用的 ID 构造一个 sparse integerized feature column，与用户的 impressioned app ID 创建的 sparse integerized feature column 进行交叉，生成 20 个 crossed feature column:

impress\_col = sparse\_integerized\_feat\_column(impressioned\_app.ID)

for i in (1,20):

wide\_col = sparse\_integerized\_feat\_column(download\_app[i].ID)

cross\_col = crossed\_column(impress\_col, wide\_col)

输入到 DNN 的连续型特征我们通过定义 real valued feature column 的 normalizer 进行 L2 归一化，normalizer 定义和使用方式如下：

def l2\_real\_val\_col\_normalizer(x)

return tf.nn.l2\_normalize(x, 0)

real\_valued\_col = tf.contrib.layers.real\_valued\_column("real\_feat\_col", normalizer = l2\_real\_val\_col\_normalizer)

L1 归一化可以定义如下 normalizer：

def l1\_real\_val\_col\_normalizer(x)

sum = math\_ops.reduce\_sum(math\_ops.abs(x), 0, keep\_dims=True)

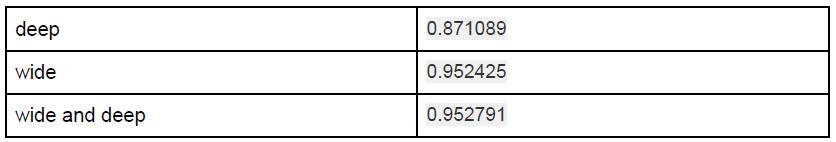
inv\_norm = 1.0 / (math\_ops.maximum(sum, epsilon))

return math\_ops.multiply(x, inv\_norm)

同时为用户下载和删除历史中的每一个应用单独构建一个 embedding column 作为 DNN 输入。

标记和特征构建好后，下一步是切分训练和测试数据集，我们按照事件的时间戳对数据按照 7：3 的比例切分成训练和测试集，并且从测试集中去掉所有在训练集中出现过的（用户，应用）对。在训练之前我们还做了 3 件事情：（1）在训练集上对部分特征做了相关性分析，看特征之间是否存在比较强的线性相关性；（2）统计训练数据上的正负样本比例，通过对正样本进行 repeated boostrap sampling 增加正样本比例。

最终模型训练中我们还尝试了：（1）enable centered bias；（2）enable layer batch normalization；（3）尝试 DNN 的其他优化函数。在我们的测试集合上模型的 AUC 如下：



模型训练评测后下一步准备上线，模型需要 export 后才能被 tensorflow serving 加载使用。目前我们没有使用上 TensorFlow1.0 以后的 export\_savedmodel，仍然使用的是 export 函数：

m.export(export\_dir=export\_dir, use\_deprecated\_input\_fn=True,signature\_fn=signature\_fn,

exports\_to\_keep=3)

export 中通过 signature\_fn 定义模型的输入和输出，signature\_fn 我们是仿照 estimator.classification\_signature\_fn\_with\_prob 进行定义的，不同点是我们增加了对 named\_graph\_sigatures 的定义：

def signature\_fn(examples, unused\_features, predictions):

if isinstance(predictions, dict):

default\_signature = exporter.classification\_signature( examples, scores\_tensor=predictions['probabilities'])

named\_graph\_signatures = {

'inputs': exporter.generic\_signature({'values': examples}),

'outputs': exporter.generic\_signature({'preds': predictions['probabilities']})}

else:

default\_signature = exporter.classification\_signature( examples, scores\_tensor=predictions)

named\_graph\_signatures = {

'inputs': exporter.generic\_signature({'values': examples}),

'outputs': exporter.generic\_signature({'preds': predictions})}

return default\_signature, named\_graph\_signatures

线上的模型服务我们启动时需要传入参数–use\_saved\_model=false   
，这样 tensorflow serving 做 prediction 时会使用 SessionBundlePredict，输入从』input』字段获取，输出从』output』字段获取，具体可看 TensorflowPredictor 的代码。业务服务用 C++编写 client 构造 PredictRequest 来访问模型服务：

using tensorflow::DataType;

using tensorflow::Example;

using tensorflow::Feature;

using tensorflow::TensorProto;

using tensorflow::serving::PredictRequest;

using tensorflow::serving::PredictResponse;

using tensorflow::serving::PredictionService;

Example example;

google::protobuf::Map<string, Feature>& feature =

\*example.mutable\_features()->mutable\_feature();

Feature feature;

feature.mutable\_float\_list()->add\_value(1.0);

feature[“key”] = feature;

string serialized\_example;

example.SerializeToString(&serialized\_example);

TensorProto tensor\_proto;

tensor\_proto.set\_dtype(DataType::DT\_STRING);

tensor\_proto.add\_string\_val(serialized\_example);

tensor\_proto.mutable\_tensor\_shape()->add\_dim()->set\_size(1);

PredictRequest request;

request.mutable\_model\_spec()->set\_name(FLAGS\_model\_spec);

(\*request.mutable\_inputs())["values"] = tensor\_proto;

PredictResponse resp;

client.Predict(request, &resp);

以上是关于 wide and deep 模型我们的分享，随着 TensorFlow 的逐渐演进，模型代码有可能会发生比较大的改变，希望借这个分享让各位读者对 TensorFlow wide and deep 模型本身的思想和 TensorFlow 的编程方式有一定了解。