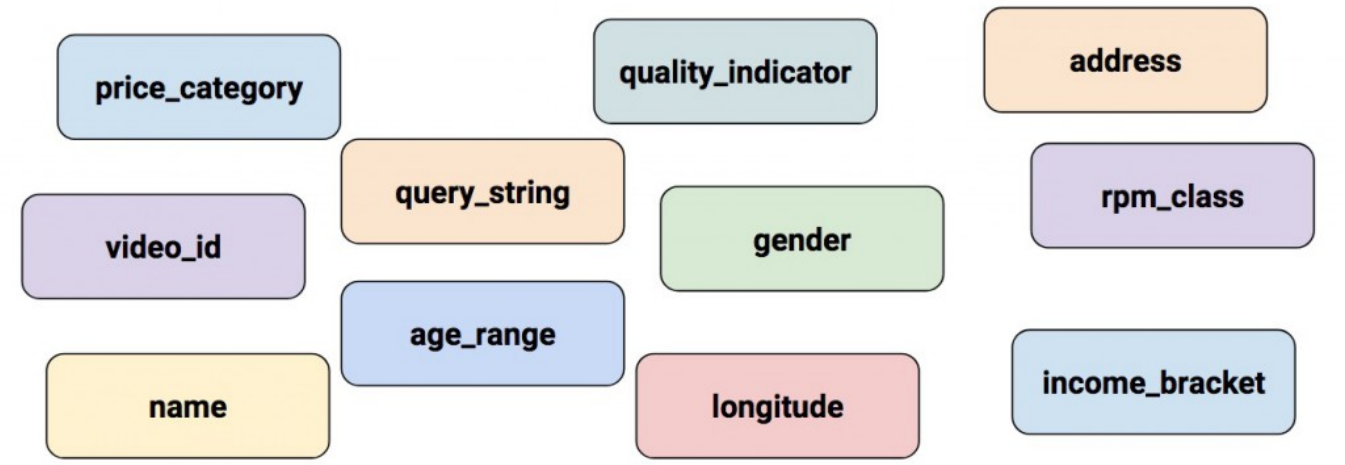

TensorFlow 特征列介绍

2017-12-11 admin GoogleDevFeeds No comments

发布人：TensorFlow 团队

欢迎阅读介绍 TensorFlow 数据集和估算器的博客系列的第 2 部分。我们将在这篇文章中介绍特征列 – 一种说明估算器进行训练和推理所需特征的数据结构。正如您将在下文看到的一样，特征列的信息非常丰富，它让您可以表示各种数据。

在第 1 部分中，我们使用了预制估算器 DNNClassifier 来训练模型，让它根据我们的输入特征预测不同类型的鸢尾花。那个示例仅创建了数值特征列（类型为 tf.feature_column.numeric_column）。尽管那些特征列足以对花瓣和萼片的长度建模，现实世界中的数据集却包含各种非数值特征。例如：

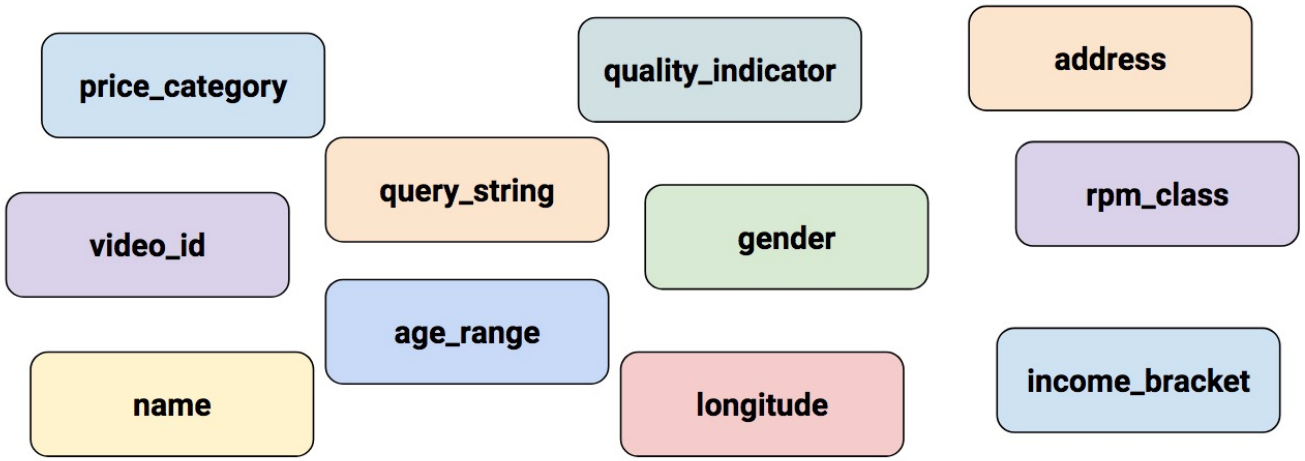

图 1.非数值特征。

怎样才能表示非数值特征类型呢？这正是我们这篇博文要讨论的内容。

深度神经网络的输入

先来问个问题：我们实际上是将哪种数据输入深度神经网络？当然，答案是数字（例如 tf.float32 ）。毕竟，神经网络中的每一个神经元都会对权重和输入数据执行乘法和加法运算。不过，现实世界中的输入数据经常包含非数值（分类）数据。例如，假设存在一个包含以下三个非数字值的 product_class 特征：

- kitchenware
- electronics
- sports

机器学习模型一般以简单矢量表示分类值，其中，1 表示某个值存在，0 表示某个值不存在。例如，当 product_class 设为 sports 时，机器学习模型通常会以 [0, 0, 1] 表示 product_class ，含义如下所示：

- 0： kitchenware 不存在
- 0： electronics 不存在
- 1： sports 存在

所以，尽管原始数据可以是数值或分类数据，机器学习模型会以数字或由数字组成的矢量表示所有特征。

特征列介绍

如图 2 中所示，您可以通过估算器（鸢尾花为 DNNClassifier ）的 feature_columns 参数指定模型的输入。特征列将输入数据（由 input_fn 返回）与您的模型联系起来。

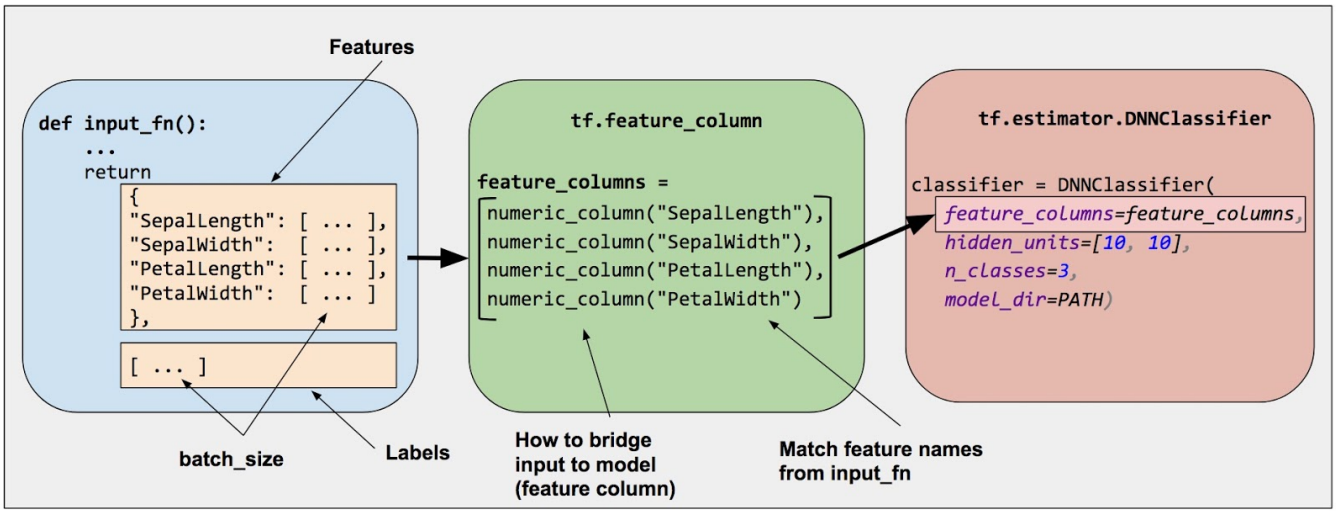

图 2.特征列将原始数据与您的模型需要的数据联系起来。

要以特征列表示特征，请调用 tf.feature_column 软件包的函数。这篇博文将介绍此软件包中的九个函数。如图 3 所示，所有九个函数都会返回一个 Categorical-Column 或 Dense-Column 对象，但 bucketized_column 除外，它继承自这两个类别：

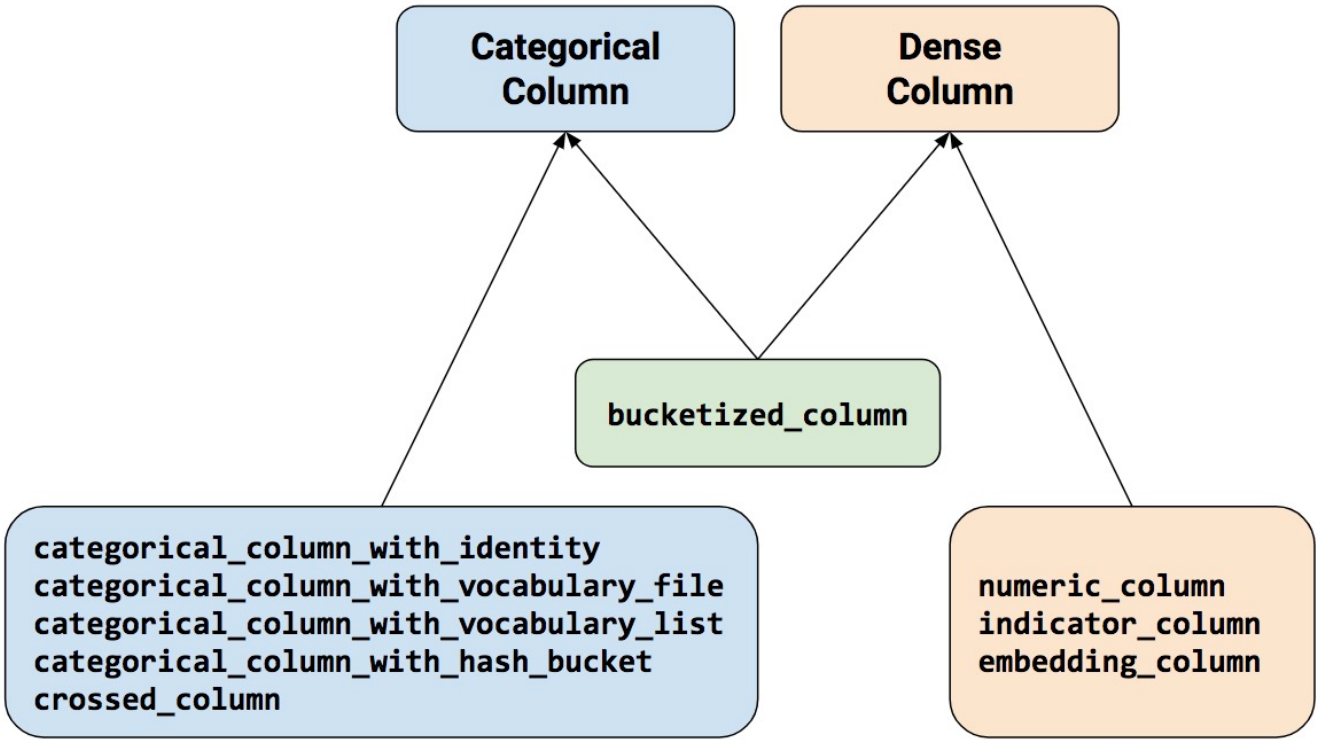

图 3.特征列函数可以归入两个主要类别和一个混合类别。

我们来详细看一下这些函数。

数值列

鸢尾花分类器为所有输入特征调用了 tf.numeric_column() ：SepalLength、SepalWidth、PetalLength、PetalWidth。尽管 tf.numeric_column() 提供了可选参数，调用不含任何参数的函数仍是指定具有默认数据类型（tf.float32）的数字值作为模型输入的一种极简单方式。例如：

```
# Defaults to a tf.float32 scalar.
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength")
```

使用 dtype 参数可以指定非默认数值数据类型。例如：

```
# Represent a tf.float64 scalar.
```

```
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength",
                                                          dtype=tf.float64)
```

默认情况下，一个数值列可以创建一个值（标量）。使用 shape 参数可以指定另一个形状。例如：

```
# Represent a 10-element vector in which each cell contains a tf.float32.
vector_feature_column = tf.feature_column.numeric_column(key="Bowling",
                                                         shape=10)

# Represent a 10x5 matrix in which each cell contains a tf.float32.
matrix_feature_column = tf.feature_column.numeric_column(key="MyMatrix",
                                                         shape=[10,5])
```

存储分区化列

通常，您不希望将数字直接提供给模型，而是根据数值范围将它的值拆分成不同的类别。为此，请创建一个存储分区化列。例如，假设存在表示房子建造年份的原始数据。我们可以将年份拆分成以下四个存储分区，而不是以标量数值列表示年份：

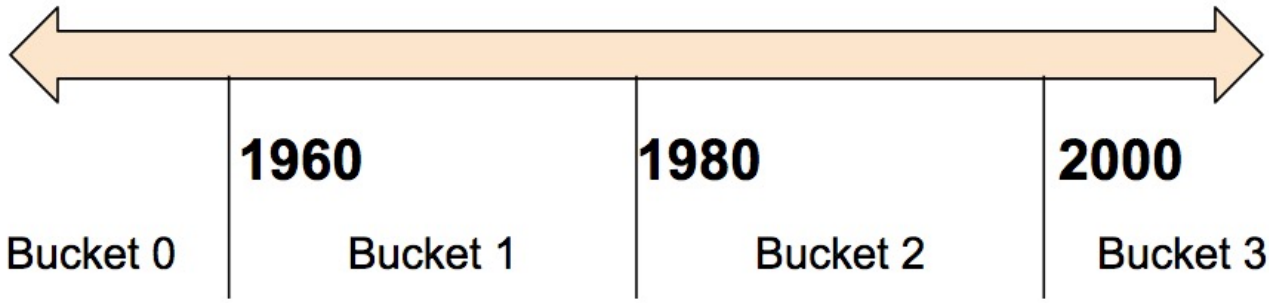

图 4.将年份数据分成四个存储分区。

模型将通过以下方式表示存储分区：

日期范围	表示形式...
< 1960	[1, 0, 0, 0]
>= 1960 且 < 1980	[0, 1, 0, 0]
>= 1980 且 < 2000	[0, 0, 1, 0]

> 2000	[0, 0, 0, 1]

既然数字对模型来说是一种非常有效的输入，为什么还要将它拆分成这样的分类值呢？请注意，分类可以将一个输入数字拆分成一个四元素矢量。因此，模型现在可以学习 *四个单独的权重* ，而不是仅仅学习一个。与一个权重相比，四个权重可以创建信息更丰富的模型。更重要的是，由于只有一个元素置位 (1)，其他三个元素清零 (0)，存储分区化可以让模型清楚地区分不同的年份类别。如果我们仅使用一个数字（年份）作为输入，模型无法区分类别。所以，存储分区化可以为模型提供可以用来学习的其他重要信息。

下面的代码演示了如何创建存储分区化特征：

```
# A numeric column for the raw input.
numeric_feature_column = tf.feature_column.numeric_column("Year")

# Bucketize the numeric column on the years 1960, 1980, and 2000
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

请注意以下事项：

- 在创建存储分区化列之前，我们先创建了一个数值列来表示原始年份。
- 我们将数值列作为第一个参数传递到 tf.feature_column.bucketized_column() 中。
- 指定一个 三元素 boundaries 矢量可以创建一个 *四*元素存储分区化矢量。

分类标识列

分类标识列是一种特殊形式的存储分区化列。在传统的存储分区化列中，每个存储分区表示一个 *范围* 的值（例如，从 1960 到 1979）。在分类标识列中，每个存储分区表示 一个唯一整数。例如，我们假设您想要表示整数范围 [0, 4)。（即，您希望表示整数 0、1、2 或 3。）在这种情况下，分类标识映射如下所示：

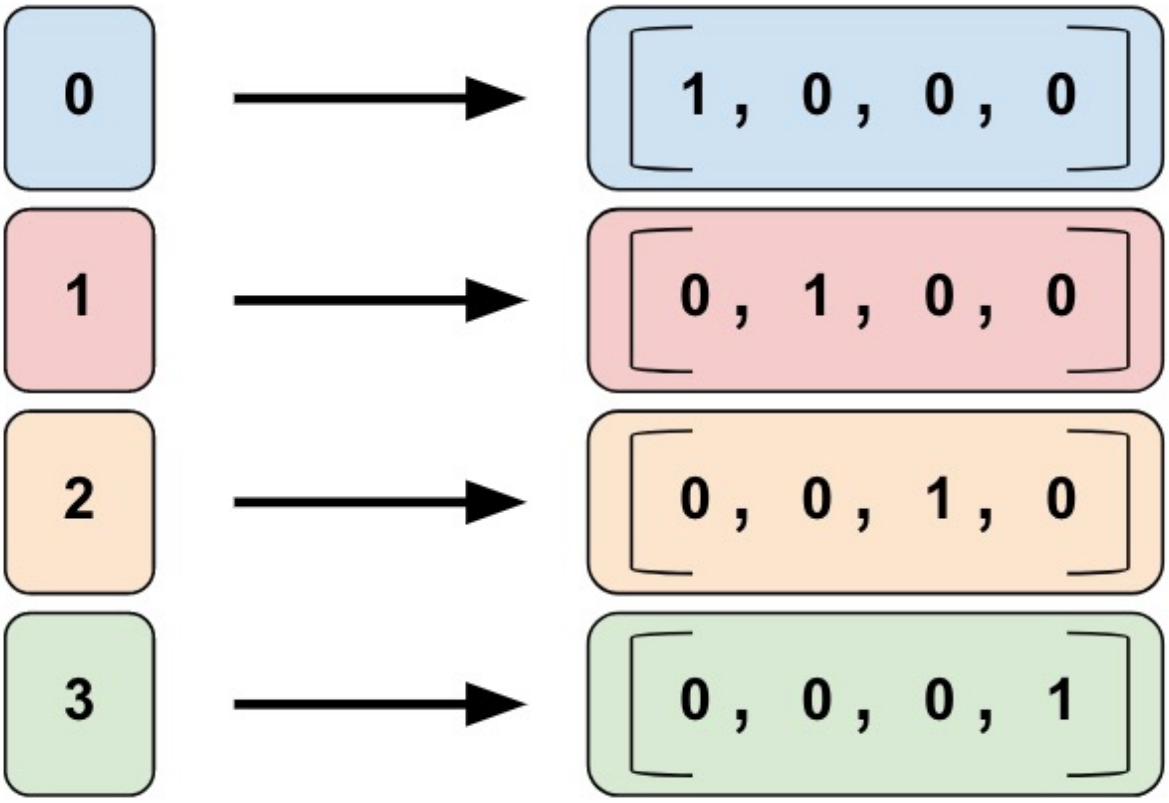

图 5.分类标识列映射。请注意，这是一种独热编码，而不是二进制数字编码。

那么，您为什么想要以分类标识列表示值呢？使用存储分区化列，模型可以在分类标识列中为每个类别学习单独的权重。例如，与使用一个字符串表示 product_class 相反，我们使用一个唯一整数值表示每个类别。即：

- 0="kitchenware"
- 1="electronics"
- 2="sport"

调用 tf.feature_column.categorical_column_with_identity() 来实现一个分类标识列。例如：

```
# Create a categorical output for input "feature_name_from_input_fn",
# which must be of integer type. Value is expected to be >= 0 and < num_buckets
identity_feature_column = tf.feature_column.categorical_column_with_identity(
```

```
    key='feature_name_from_input_fn',
    num_buckets=4) # Values [0, 4)

# The 'feature_name_from_input_fn' above needs to match an integer key that is
# returned from input_fn (see below). So for this case, 'Integer_1' or
# 'Integer_2' would be valid strings instead of 'feature_name_from_input_fn'.
# For more information, please check out Part 1 of this blog series.
def input_fn():
    ...<code>...
    return ({ 'Integer_1':[values], ..<etc>.., 'Integer_2':[values] },
            [Label_values])
```

分类词汇列

我们无法将字符串直接输入模型。相反，我们必须先将字符串映射到数字或分类值。分类词汇列提供了一种以独热矢量表示字符串的好方法。例如：

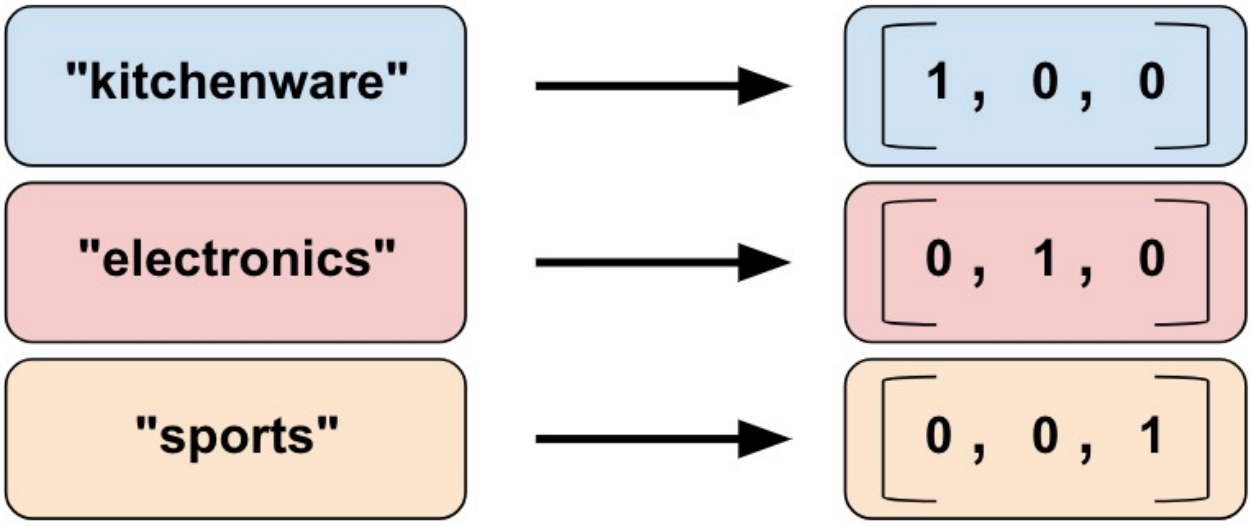

图 6.将字符串值映射到词汇列。

如您所见，分类词汇列是一种枚举版本的分类标识列。TensorFlow 提供了两个不同的函数来创建分类词汇列：

- tf.feature_column.categorical_column_with_vocabulary_list()

- tf.feature_column.categorical_column_with_vocabulary_file()

tf.feature_column.categorical_column_with_vocabulary_list() 函数可以根据一个显式词汇列表将每个字符串映射到一个整数。例如：

```
# Given input "feature_name_from_input_fn" which is a string,
# create a categorical feature to our model by mapping the input to one of
# the elements in the vocabulary list.
vocabulary_feature_column =
  tf.feature_column.categorical_column_with_vocabulary_list(
    key="feature_name_from_input_fn",
    vocabulary_list=["kitchenware", "electronics", "sports"])
```

前面的函数有一个明显的缺陷；也就是说，当词汇列表较长时，需要进行很多输入工作。对于这些情况，请改为调用 tf.feature_column.categorical_column_with_vocabulary_file() ，它让您可以将词汇放在一个单独的文件中。例如：

```
# Given input "feature_name_from_input_fn" which is a string,
# create a categorical feature to our model by mapping the input to one of
# the elements in the vocabulary file
vocabulary_feature_column =
  tf.feature_column.categorical_column_with_vocabulary_file(
    key="feature_name_from_input_fn",
    vocabulary_file="product_class.txt",
    vocabulary_size=3)

# product_class.txt should have one line for vocabulary element, in our case:
kitchenware
electronics
sports
```

使用哈希存储分区限制类别

到目前为止，我们仅介绍了少量几个类别。例如，我们的 product_class 示例仅有 3 个类别。但是，类别的数量通常可以非常大，以致于无法为每个词汇或整数使用单独的类别，因为这样会消耗大量内存。对于这些情况，我们可以将问题反过来并问问自己，“我愿意为输入使用多少个类别？”事实上，tf.feature_column.categorical_column_with_hash_buckets() 函数让您可以指定类别数量。例如，以下代码

显示了此函数如何计算输入的哈希值，然后使用模数运算符将其置于一个 hash_bucket_size 类别中：

```
# Create categorical output for input "feature_name_from_input_fn".
# Category becomes: hash_value("feature_name_from_input_fn") % hash_bucket_size
hashed_feature_column =
  tf.feature_column.categorical_column_with_hash_bucket(
    key = "feature_name_from_input_fn",
    hash_buckets_size = 100) # The number of categories
```

此时，您可能会想：“这太疯狂了！”毕竟，我们在将不同的输入值强制到一组较小的类别中。这意味着，两个很可能完全不相关的输入将被映射到同一个类别，这对神经网络来说也是一样的。图 7 说明了这个难题，显示 kitchenware 和 sports 都分配获得了类别（哈希存储分区）12：

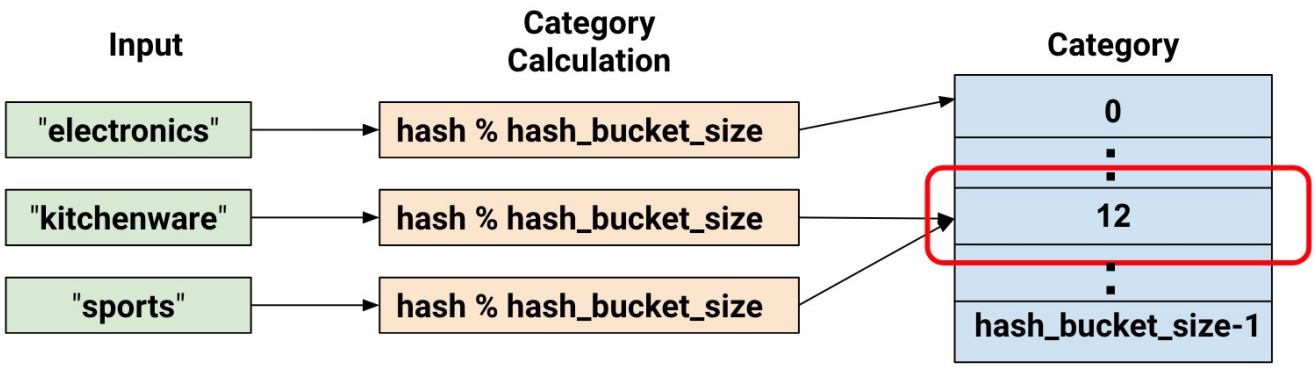

图 7.在哈希存储分区中表示数据。

与机器学习中许多有悖常理的现象一样，哈希通常可以在实践中很好地运行。这是因为哈希类别为模型提供了一些分隔。模型可以使用更多特征来进一步将 kitchenware 与 sports 分开。

特征交叉

我们要介绍的最后一个分类列允许我们将多个输入特征组合成一个。组合特征（更广为人知的说法是特征交叉）让模型可以专门针对特征组合表示的任何意义学习单独的权重。

更具体一点，假设我们希望我们的模型计算佐治亚州亚特兰大的房地产价格。这个城市的房地产价格因位置不同而相差很大。以单独的特征表示纬度和经度在确定房地产位置相关性中不是很有用；不过，将纬度和经度组合到一个特征中可以确定位置。假设我们以一个 100×100 大小的矩形剖面网格表示亚特兰大，通过纬度和经

度交叉来确定 10,000 个剖面。这个组合让模型可以拾取与各个剖面相关的定价条件，与单独的纬度和经度相比，这样可以提供更强大的信息。

图 8 显示了我们的平面图，其中包含城市四个角落的纬度和经度值：

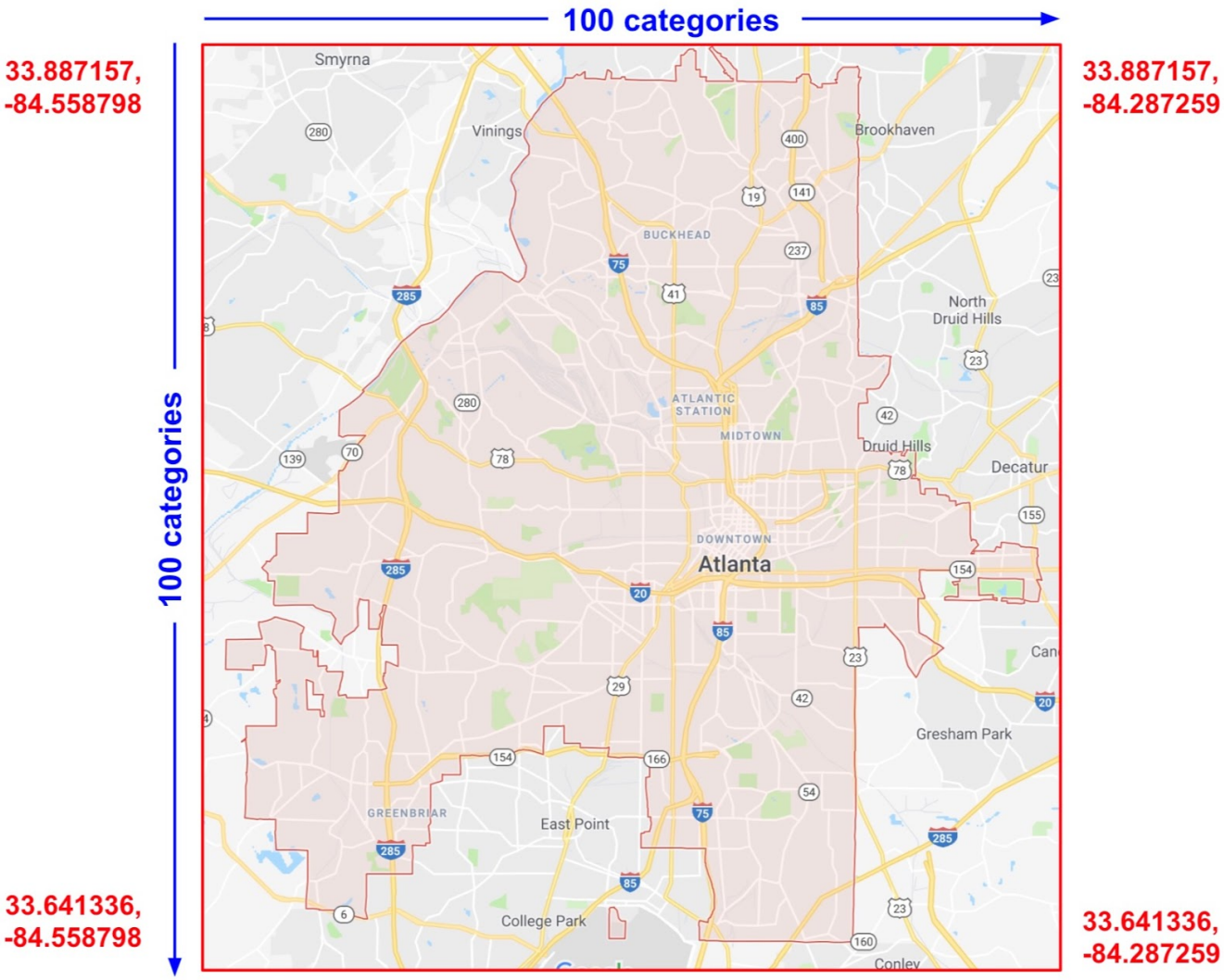

图 8.亚特兰大地图。将此地图想象成由 10,000 个大小相等的剖面组成。

为了解决问题，我们使用了之前介绍过的一些特征列组合，以及 tf.feature_columns.crossed_column() 函数。

```
# In our input_fn, we convert input longitude and latitude to integer values
```

```
# in the range [0, 100)
def input_fn():
    # Using Datasets, read the input values for longitude and latitude
    latitude = ...   # A tf.float32 value
    longitude = ...  # A tf.float32 value

    # In our example we just return our lat_int, long_int features.
    # The dictionary of a complete program would probably have more keys.
    return { "latitude": latitude, "longitude": longitude, ...}, labels

# As can be see from the map, we want to split the latitude range
# [33.641336, 33.887157] into 100 buckets. To do this we use np.linspace
# to get a list of 99 numbers between min and max of this range.
# Using this list we can bucketize latitude into 100 buckets.
latitude_buckets = list(np.linspace(33.641336, 33.887157, 99))
latitude_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('latitude'),
    latitude_buckets)

# Do the same bucketization for longitude as done for latitude.
longitude_buckets = list(np.linspace(-84.558798, -84.287259, 99))
longitude_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('longitude'), longitude_buckets)

# Create a feature cross of fc_longitude x fc_latitude.
fc_san_francisco_boxed = tf.feature_column.crossed_column(
    keys=[latitude_fc, longitude_fc],
    hash_bucket_size=1000) # No precise rule, maybe 1000 buckets will be good?
```

您可以从以下信息之一创建特征交叉：

- 特征名称；即从 input_fn 返回的 dict 中的名称。
- 任何分类列（参见图 3），除 categorical_column_with_hash_bucket 外。

当特征列 latitude_fc 和 longitude_fc 交叉时，TensorFlow 将创建 10,000 个按以下方式组织的 (latitude_fc， longitude_fc) 组合：

```
(0,0),(0,1)...  (0,99)
(1,0),(1,1)...  (1,99)
..., ...,       ...
```

(99,0),(99,1)...(99, 99)

函数 tf.feature_column.crossed_column 将在这些组合上执行哈希计算，然后通过使用 hash_bucket_size 执行模数运算，将结果插入类别中。如之前的讨论一样，执行哈希和模数函数很可能会导致类别冲突；即多个 (纬度, 经度) 特征交叉将出现在同一个哈希存储分区中。不过在实践中，执行特征交叉仍可以为模型的学习能力提供有效值。

有点违反常理的是，在创建特征交叉时，您通常仍需要在模型中包含原始（非交叉）特征。例如，不仅提供 (latitude, longitude) 特征交叉，还需要以单独的特征形式提供 latitude 和 longitude 。单独的 latitude 和 longitude 特征将帮助模型分隔包含不同特征交叉的哈希存储分区的内容。

有关完整的代码示例，请参阅此链接。此外，此博文结尾的“参考资源”部分提供了特征交叉的更多示例。

指示器列和嵌入列

指示器列和嵌入列永远不会直接在特征上运行，而是将分类列作为输入。

使用指示器列时，我们将告知 TensorFlow 准确执行我们在分类 product_class 示例中看到的操作。即，指示器列将每个类别作为独热矢量中的一个元素处理，其中，匹配类别的值为 1，其余为 0：

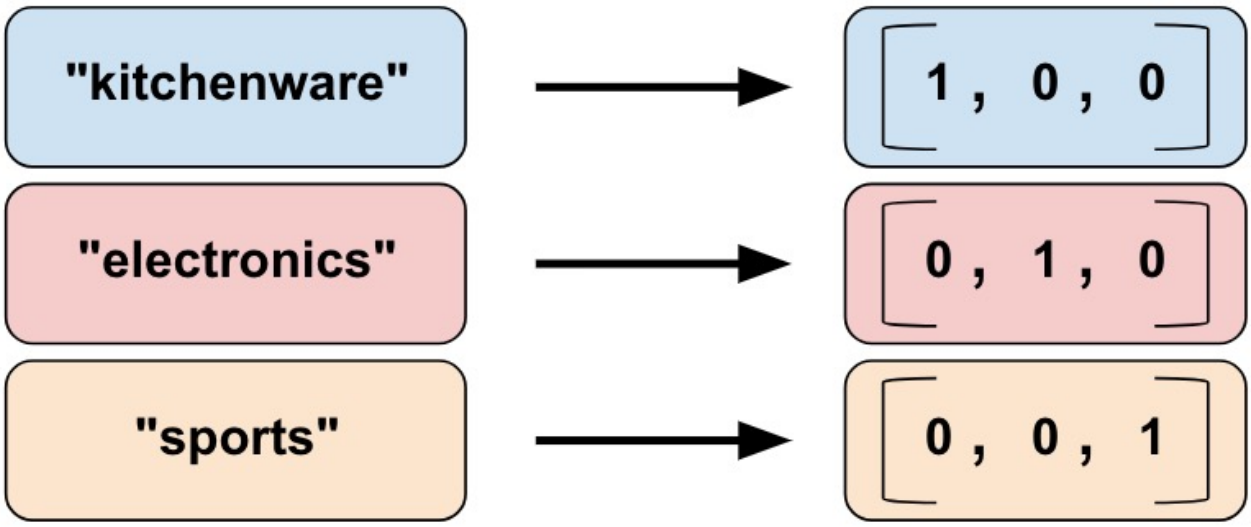

图 9.在指示器列中表示数据。

下面是创建指示器列的方式：

```
categorical_column = ... # Create any type of categorical column, see Figure 3

# Represent the categorical column as an indicator column.
# This means creating a one-hot vector with one element for each category.
indicator_column = tf.feature_column.indicator_column(categorical_column)
```

现在，假设我们不只有三个可能的类别，而是有 100 万个。或者有 10 亿个。由于多种原因（技术性过强，我们无法在此介绍），在类别数量增大时，使用指示器列训练神经网络将变得不可行。

我们可以使用嵌入列来克服此限制。与将数据表示为具有许多维度的独热矢量不同，嵌入列将数据表示为一个更低维度的普通矢量，其中，每个单元格都可以包含任意数字，而不仅仅是 0 或 1。通过为每个单元格允许更丰富的数字组合，与指示器列相比，嵌入列可以包含少得多的单元格数。

我们来看一个比较指示器列和嵌入列的示例。假设我们的输入示例包含一个有限组合中的不同单词，这个组合仅有 81 个单词。再假设数据集在 4 个单独的示例中提供以下输入单词：

- “dog”
- “spoon”
- “scissors”
- “guitar”

在这种情况下，图 10 说明了嵌入列或指示器列的处理路径。

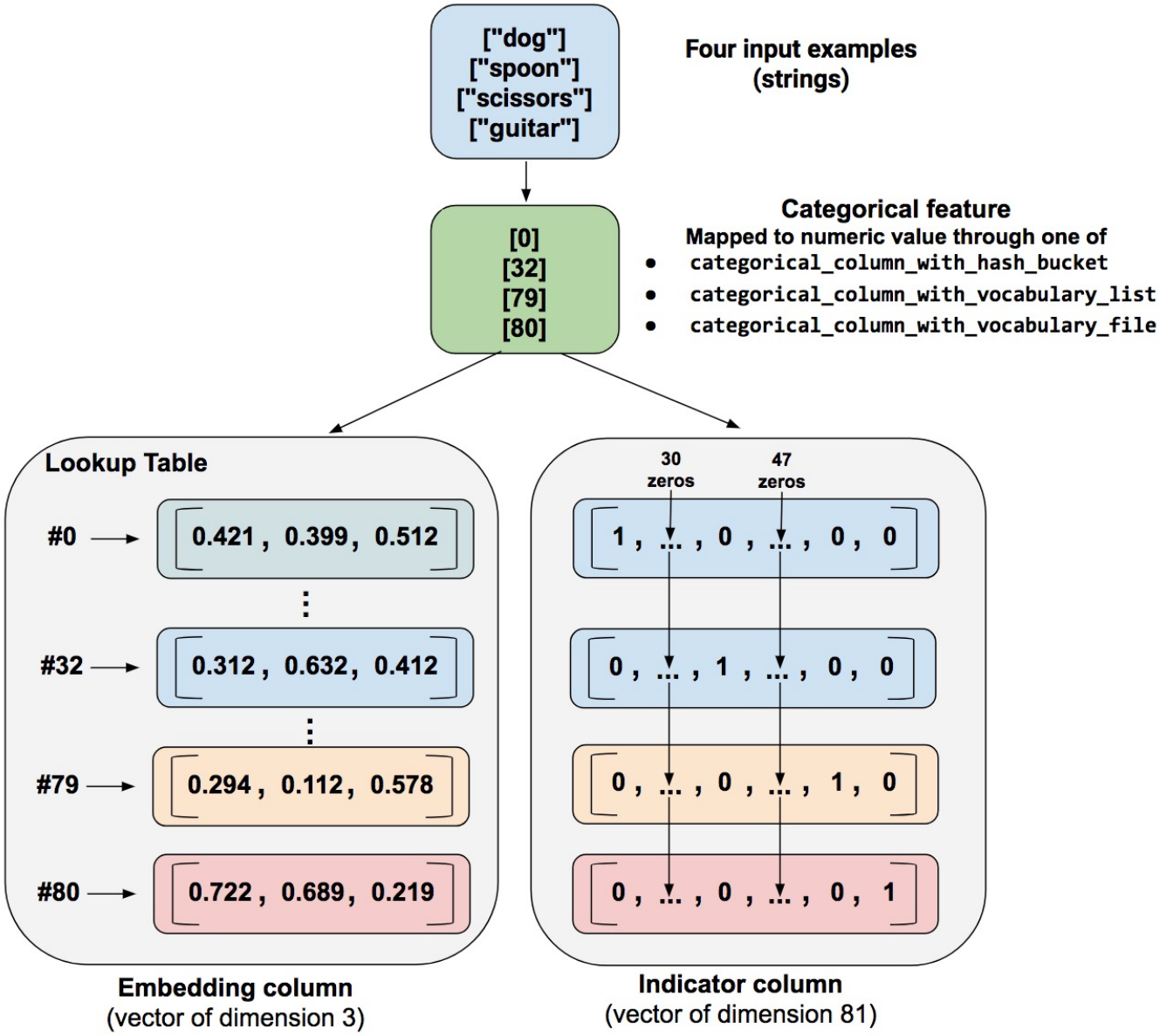

图 10.与指示器列相比，嵌入列会将分类数据存储在一个更低维度的矢量中。（我们仅将随机数字置于嵌入矢量中；训练决定实际数字。）

在处理一个示例时，一个 categorical_column_with... 函数会将示例字符串映射到数字分类值。例如，一个函数会将 "spoon" 映射到 [32]。（ 32 来自我们的想象 – 实际值取决于映射函数。）然后，您可以通过以下两种方式之一表示这些数字分类值：

- 作为指示器列。函数会将每个数字分类值转换成一个 81 元素矢量（因为我们的组合包含 81 个单词），

在分类值 (0, 32, 79, 80) 的索引中放置 1，在其他位置放置 0。

- 作为嵌入列。一个函数使用数字分类值 (0, 32, 79, 80) 作为查找表的索引。查找表中的每个显示位置都包含一个 3 元素矢量。

嵌入矢量中的值是怎样魔术般地获得分配的？实际上，分配发生在训练期间。即，模型将学习最佳方式来将您的输入数字分类值映射到嵌入矢量值，求解您的问题。嵌入列可以提升您的模型的能力，因为嵌入矢量可以从训练数据中学习类别之间的新关系。

我们示例中矢量的大小为什么是 3？以下“公式”提供了与嵌入维度数量有关的一般经验法则：

```
embedding_dimensions =  number_of_categories**0.25
```

即，嵌入矢量维度应等于类别数量的四次方根。由于此示例中的词汇大小为 81，建议的维度数量为 3：

```
3 =  81**0.25
```

请注意，这只是一个一般准则；您可以随意设置嵌入维度的数量。

调用 tf.feature_column.embedding_column 来创建一个 embedding_column。嵌入矢量的维度取决于上面介绍的手头问题，但常用值可以从最低的 3 开始，一直到 300 或更大：

```
categorical_column = ... # Create any categorical column shown in Figure 3.

# Represent the categorical column as an embedding column.
# This means creating a one-hot vector with one element for each category.
embedding_column = tf.feature_column.embedding_column(
    categorical_column=categorical_column,
    dimension=dimension_of_embedding_vector)
```

嵌入是机器学习领域的一个大主题。此信息旨在帮助您开始将它们用作特征列。请参阅此博文的结尾了解更多信息。

将特征列传递到估算器

还在看吗？我希望大家还在看，因为特征列的基础知识马上就要介绍完了。

正如我们在图 1 中看到的一样，特征列可以将您的输入数据（通过从 input_fn 返回的特征字典进行说明）映射到要提供给模型的值。以列表形式将特征列指定到估算器的 feature_columns 参数。请注意，feature_columns 参数因估算器的不同而有所差异：

- LinearClassifier 和 LinearRegressor ：
  - 接受所有类型的特征列。
- DNNClassifier 和 DNNRegressor：
  - 仅接受密集列，请参见图 3。如之前所述，其他列类型必须包装在 indicator_column 或 embedding_column 中。
- DNNLinearCombinedClassifier 和 DNNLinearCombinedRegressor ：
  - linear_feature_columns 参数可以接受任意列类型，像上面的 LinearClassifier 和 LinearRegressor 一样。
  - 不过， dnn_feature_columns 参数被限制为密集列，像上面的 DNNClassifier 和 DNNRegressor 一样。

上述规则的原因超出了这篇介绍博文的范围，不过，我们会在未来的博文中确保对此进行介绍。

总结

使用特征列可以将您的输入数据映射到您向模型提供的表示。我们在此系列的第 1 部分中仅使用了 numeric_column ，不过使用这篇博文介绍的其他函数，您可以轻松创建其他特征列。

如需了解有关特征列的更多详细信息，请参阅下面的资源：

- Josh Gordon 介绍特征工程的视频
- 来自同一位作者的 Jupyter 笔记
- TensorFlow – Wide & Deep 教程
- DNN 和使用特征列的线性模型的示例

如果您想详细了解嵌入，请参阅以下资源：

- 深度学习、NLP 和表示（Colah 的博客）
- 参阅 TensorFlow Embedding Projector

.blgimg1 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg2 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg3 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg4 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg5 img { width:50%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg6 img { width: 50%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg7 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg8 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg9 img { width: 50%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } .blgimg10 img { width: 100%; border: 0; margin: 0; padding: 10px 0 10px 0 ; } code { font-size: 100%; } table, th { border: 1px solid black; border-collapse: collapse; } td { border: 1px solid black; border-collapse: collapse; width: 50% !important; } <!—->

Source: TensorFlow 特征列介绍

Develop

Related Articles

SafetyNet 认证，反滥用的构建基块 2017-05-05

Some Tips for Boosting your App's Quality in 2017 2017-01-05

推出面向网站开发者的 Mobile Sites 认证 2017-04-14

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment *

POST COMMENT

search ...

Recent Posts

- Introducing NIMA: Neural Image Assessment
- Quick Boot & the Top Features in the Android Emulator
- What a year! Google Cloud Platform in 2017
- Cloud Audit Logging for Kubernetes Engine: Answer the who, what, when of admin accesses
- 让 Pixel 更适合驾驶员

Recent Comments

- 王中 on Google 推出的 31 套在线课程
- Francis Wang on Google 推出的 31 套在线课程
- daruo on 面向普通开发者的机器学习应用方案
- Hmark on Google 推出的 31 套在线课程
- Dambo on Open sourcing the Firebase SDKs

Archives

- December 2017
- November 2017
- October 2017
- September 2017
- August 2017
- July 2017
- June 2017
- May 2017
- April 2017
- March 2017
- February 2017
- January 2017
- December 2016
- November 2016
- October 2016
- September 2016

- August 2016
- May 2016
- April 2016
- March 2016
- February 2016
- January 2016
- December 2015
- November 2015
- October 2015
- September 2015
- August 2015
- July 2015
- June 2015
- January 1970

Categories

- Android
- Design
- Firebase
- GoogleCloud
- GoogleDevFeeds
- GoogleMaps
- GooglePlay

- Google动态
- iOS
- Uncategorized
- VR
- Web
- WebMaster
- 社区
- 通知

Meta

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.org