

# 手把手教你为iOS系统开发TensorFlow应用



机器之心 企鹅号

原创 2017-04-12

选自machinethink.net

参与：赵华龙、邵明、吴攀、李泽南

在你使用深度神经网络做预测之前，你首先要训练神经网络。现在存在许多不同的神经网络训练工具，TensorFlow 正迅速成为其中最热门的选择。近日，独立开发者 Matthijs Hollemans 在 machinethink.net 的博客上发布了一篇讲解如何在 iOS 系统上运行 TensorFlow 的深度长文教程，并开源了相关的代码。机器之心对本文进行了编译介绍。关于 TensorFlow 的更多资讯和教程，请参阅机器之心文章《谷歌召开首届 TensorFlow 开发者大会，正式发布 TensorFlow 1.0》和《首届 TensorFlow 开发者大会：值得关注的亮点都在这里（附资源）》。

项目地址：<https://github.com/hollance/TensorFlow-iOS-Example>

你可以使用 TensorFlow 来训练你的机器学习模型，并使用这些模型进行预测。训练通常在强大的机器上或云端完成，但是 TensorFlow 也可以在 iOS 上运行——尽管有一些限制。

在这篇博文中，我将解释 TensorFlow 背后的思想，如何使用它来训练一个简单的分类器，以及如何将这个分类器放在你的 iOS 应用程序中。

我们将通过音频和语音分析的性别识别数据集（Gender Recognition by Voice and Speech Analysis, <https://www.kaggle.com/primaryobjects/voicegender>）来学习如何将一段录音识别为男性或女性的声音。你可以在上面的 GitHub 地址找到此项目的源代码。

## 什么是 TensorFlow 以及为何我需要它？

TensorFlow 是一个用于构建计算图（computational graph）以便进行机器学习的软件库。

许多其它的工具工作在更高的抽象层次上。以 Caffe 为例，你可以通过连接不同类型的「层（layer）」来设计神经网络。这和 iOS 中 BNNS 以及 MPSCNN 的功能类似。在 TensorFlow 中，你也可以使用这样的层来工作，不过你还可以做得更深入，一直到构成你算法的单个计算。

你可以将 TensorFlow 视为一个实现新机器学习算法的工具包，而其它的深度学习工具则是为了使用其他人实现的那些算法。

这并不意味着你总是要从头开始构建一切。TensorFlow 附带有可复用的构建块的集合，而且还有其它库（如 Keras）也在 TensorFlow 上提供了方便的模块。

所以精通数学不是使用 TensorFlow 的一个要求，但如果你想成为顶尖专家，还是应该掌握。

## 基于 logistic 回归的二值分类

在这篇博文中，我们将使用 logistic 回归算法创建一个分类器。是的，我们将从头开始构建它。那就卷起袖子开始吧！

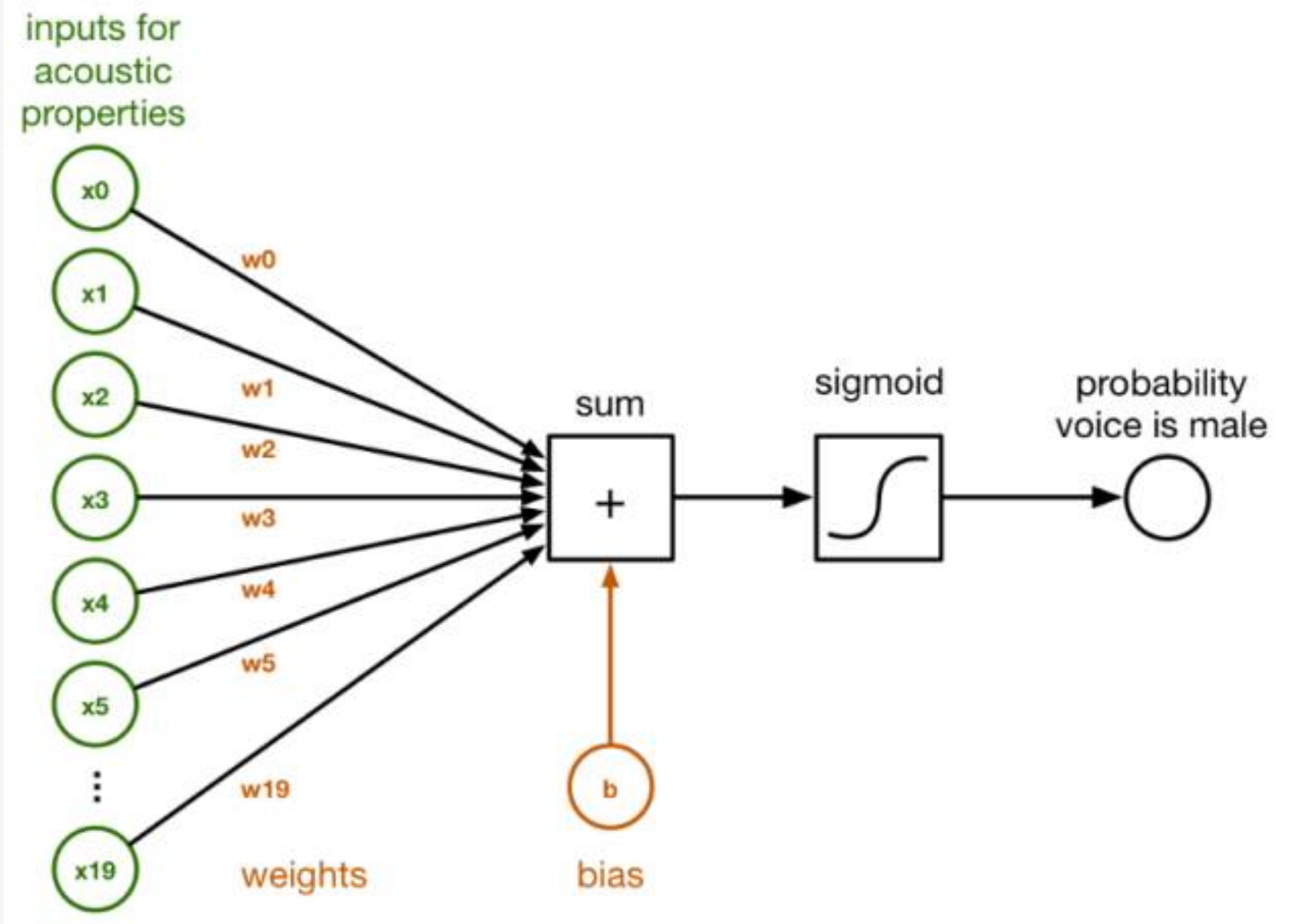
分类器接收一些输入数据，然后告诉你该数据属于哪个类别或类。对于这个项目，我们只有两个类：男性或女性，因此我们是一个二值分类器（binary classifier）。

注：二值分类器是最简单的分类器，但它使用的思路与可以区分数百或数千个不同的类的分类器相同。所以即使我们在本教程中并没有做到真正的深度学习，但两者也仍有很多共同之处。

我们将使用的输入数据由 20 个数字组成，这些数字代表某人说话的特定录音的各种声学特性。稍后我会再解释一下，其中包括音频的频率这样的信息。

在下图中，你可以看到这 20 个数字连接到一个叫做 sum 的小块。根据分类器，这些连接具有不同的权重（weight），这对应于这 20 个数字中的每一个的重要程度。

这是 logistic 分类器工作方式的框图：



在 sum 块内，由  $x_0 - x_{19}$  给出的输入及其连接权重  $w_0 - w_{19}$  的权重被简单的加在一起。这就是一个常规的点积：

我们还在最后添加了一个所谓的偏置（bias）项  $b$ 。这只是另一个数字。

数组中的权重  $w$  和  $b$  的值是该分类器要学习的内容。训练分类器就是找到  $w$  和  $b$  的正确数字的问题。最初，我们将所有的  $w$  和  $b$  设为零。经过多轮训练， $w$  和  $b$  将包含一组数字，分类器可以用它们来分辨男性的语音和女性的语音。

为了将这个 sum 值转换成 0 到 1 之间的概率值，我们采用 logistic sigmoid 函数：

这个公式可能看起来很吓人，但它的作用很简单：如果 sum 是一个大的正数，则该 sigmoid 函数返回 1，表示概率为 100%。如果 sum 是一个大的负数，则该 sigmoid 函数返回 0。所以对于大的正数或负数，我们可以得到一个有信心的「是」或「否」预测。

然而，如果 sum 接近 0，则该 sigmoid 函数给出的概率接近 50%，因为它对预测不能确信。当我们开始训练该分类器时，初始预测将对半分，因为分类器还没有学到任何东西，并

且对结果没有信心。但是我们训练越多，概率越趋于 1 和 0，分类器变得越明确。

现在，`y_pred` 包含了预测结果，其概率表示的是该语音来自男性的概率。即如果超过 0.5（或 50%），我们认为该声音是男性，否则是女性。

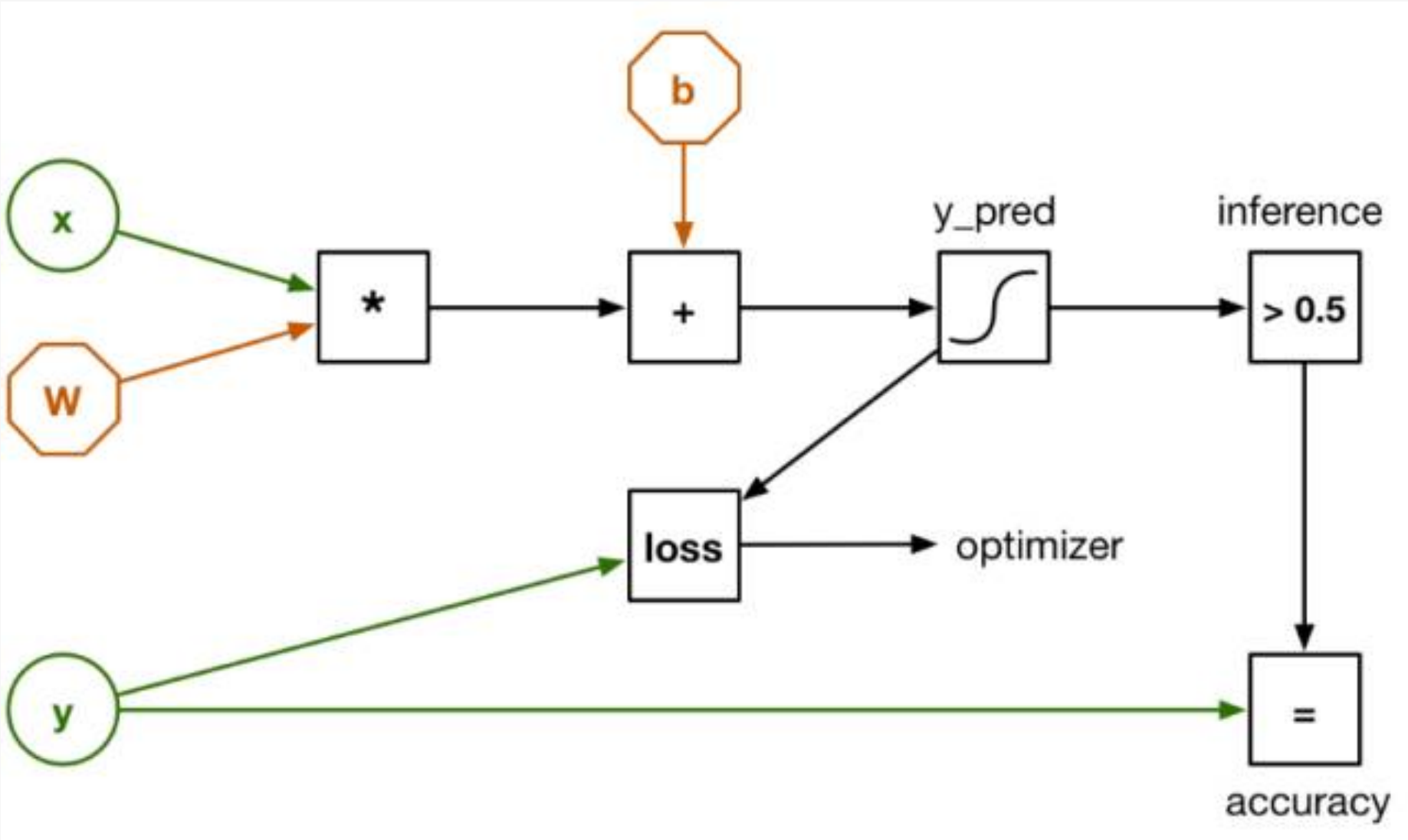
这就是使用 logistic 回归进行二值分类的思想。分类器的输入由描述音频录音的声学特征的 20 个数字组成，我们计算加权和并采用 sigmoid 函数，我们得到的输出是说话人是男性的概率。

然而，我们仍然需要建立训练该分类器的机制，所以现在来看看 TensorFlow。

### 在 TensorFlow 中实现该分类器

要在 TensorFlow 中使用该分类器，我们首先需要将其设计转换为计算图（computational graph）。计算图由执行计算的节点和在这些节点之间流动的数据组成。

我们的 logistic 回归的图如下所示：



这与前面的图表看起来有点不同，但是这是因为输入 `x` 不再是 20 个独立的数字，而是包含 20 个元素的向量。权重由矩阵 `W` 表示。因此，之前的点积被单个矩阵乘法替代了。

还有一个输入  $y$ 。这用于训练分类器并验证其有效性。我们使用的数据集有 3,168 个录音样本，每个样本我们也知道是男声还是女声。那些已知的结果（男性或女性）也被称为数据集的标签（label），而这就是我们将放在  $y$  中的。

为了训练该分类器，我们将其中一个样本加载到  $x$  中，并让该图做出预测：是男性还是女性？因为最初的权重都是零，所以分类器可能会做出错误的预测。我们需要一种方法来计算错误的程度——通过损失函数（loss function）。损失函数将预测结果  $y_{\text{pred}}$  与正确的结果  $y$  进行比较。

确定了训练样本的损失后，我们使用一种称为反向传播（backpropagation）的技术，反向通过计算图来调整权重  $W$  和  $b$  到正确的方向。如果预测是男性，但正确的答案是女性，权重就会上下移动一点，使得下一次「女性」将更有可能成为该特定的输入的结果。

该训练过程在该数据集的所有样本上一次又一次地重复，直到该图确定了最佳权重集。随着时间的推移，用来衡量预测错误的损失就会变得越来越低。

反向传播是训练这种计算图的一种很好的技术，但是所涉及的数学可能有点棘手。而 TensorFlow 的炫酷之处就在于：由于我们将所有的「前向」操作都表示为了图中的节点，因此其可以自动找出反向传播的「反向」操作——你自己不必进行任何数学运算，很赞吧！

## 那么什么是张量呢？

在上面的图中，数据从左到右流动，从输入到输出。这就是 TensorFlow 名称中的「流（flow）」部分。但什么是「张量（tensor）」呢？

流经图的所有数据都是以张量的形式存在的。张量只是一个  $n$  维数组的一个很酷的名字。我说过  $W$  是权重矩阵，但就 TensorFlow 而言，它实际上是一个二阶张量（second-order tensor）——换句话说就是一个二维数组。

标量数是零阶张量

向量是一阶张量

矩阵是二阶张量

三维数组是一个三阶张量

等.....



这就是需要说明的一切。在卷积神经网络等深度学习方法中，你通常需要处理四维的张量，但是我们的 logistic 分类器要简单得多，所以我们不会超出二阶张量，或者说矩阵。

我还说过  $x$  是一个向量——或者说一个一阶张量——但是我们将把它当作一个矩阵。同样  $y$  也是如此。这使我们可以一次性计算整个数据集的损失。

单个样本有 20 个数据元素。如果我们将所有 3,168 个样本加载到  $x$  中，则  $x$  将成为一个  $3168 \times 20$  矩阵。将  $x$  与  $W$  相乘后，结果  $y\_pred$  为一个  $3168 \times 1$  矩阵。也就是说， $y\_pred$  对于数据集中的每个样本都有一个预测。

通过用矩阵/张量表达我们的图，我们可以一次性对许多样本案例进行预测。

## 安装 TensorFlow

好的，理论结束。现在让我们把它付诸实践。我们将使用 TensorFlow 与 Python。你的 Mac 可能已经安装了一个 Python 版本，但它可能是一个旧版本。我在本教程中使用了 Python 3.6，所以最好你也安装它。

使用 Homebrew 软件包管理器安装 Python 3.6 是最简单的。如果你还没有安装 Homebrew 软件，请先按照这些说明进行操作：<https://brew.sh>

然后打开一个终端，并键入以下命令来安装最新版本的 Python：

Python 自带它自己的软件包管理器 pip，我们将使用它来安装我们需要的软件包。从终端里，执行以下操作：

```
pip3 install numpy
pip3 install scipy
pip3 install scikit-learn
pip3 install pandas
pip3 install tensorflow
```

除了 TensorFlow，我们还将安装 NumPy、SciPy、pandas 和 scikit-learn：

NumPy 是一个使用  $n$  维数组的库。听起来很熟悉么？NumPy 不称它们为张量，但是它们是一回事。TensorFlow Python API 构建在 NumPy 之上。

SciPy 是数值计算库。它被其它一些软件包所使用。

pandas 用于加载数据集并清理它们。

scikit-learn 在某种程度上是 TensorFlow 的竞争对手，因为它是一个机器学习库。我们在我们的项目里使用它是因为它有一些方便的函数。由于 TensorFlow 和 scikit-learn 都使用 NumPy 数组，因此它们可以很好地协同工作。

你使用 TensorFlow 不一定需要 pandas 和 scikit-learn，但它们方便而且任何数据科学家的工具箱都可以有它。

这些包将被安装在 `/usr/local/lib/python3.6/site-packages`。这在如果你需要查看 TensorFlow 源代码而网站上没有相关文档的情况下是很有用的。

注：pip 将为你的系统自动安装最好的 TensorFlow 版本。如果要安装其他版本，请参阅官方安装说明 (<https://www.tensorflow.org/install>)。你还可以从源代码编译 TensorFlow，当我们构建适用于 iOS 的 TensorFlow 时，我们会在稍后做一些这样的事。

让我们进行一个快速测试，以确保一切安装正确。创建一个新的包含以下内容的文本文件 `tryit.py`：

```
pip3 install numpy
pip3 install scipy
pip3 install scikit-learn
pip3 install pandas
pip3 install tensorflow
```

然后从终端运行此脚本：

它将输出有关 TensorFlow 正在运行的设备的一些调试信息（很可能是 CPU，但也可能是 GPU——如果您的 Mac 具有 NVIDIA GPU）。最后它应该输出，

这是两个向量 `a` 和 `b` 的和。

你可能还会看到以下消息：

```
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
```

如果发生这种情况，那就说明你的系统上安装的 TensorFlow 版本对你的 CPU 而言不是最为合适的。解决此问题的一种方法是从源代码编译 TensorFlow：[https://www.tensorflow.org/install/install\\_sources](https://www.tensorflow.org/install/install_sources)，因为这样你可以配置所有选项。但到目前

来说，如果你看到这些警告，也并不是什么大不了的事情。

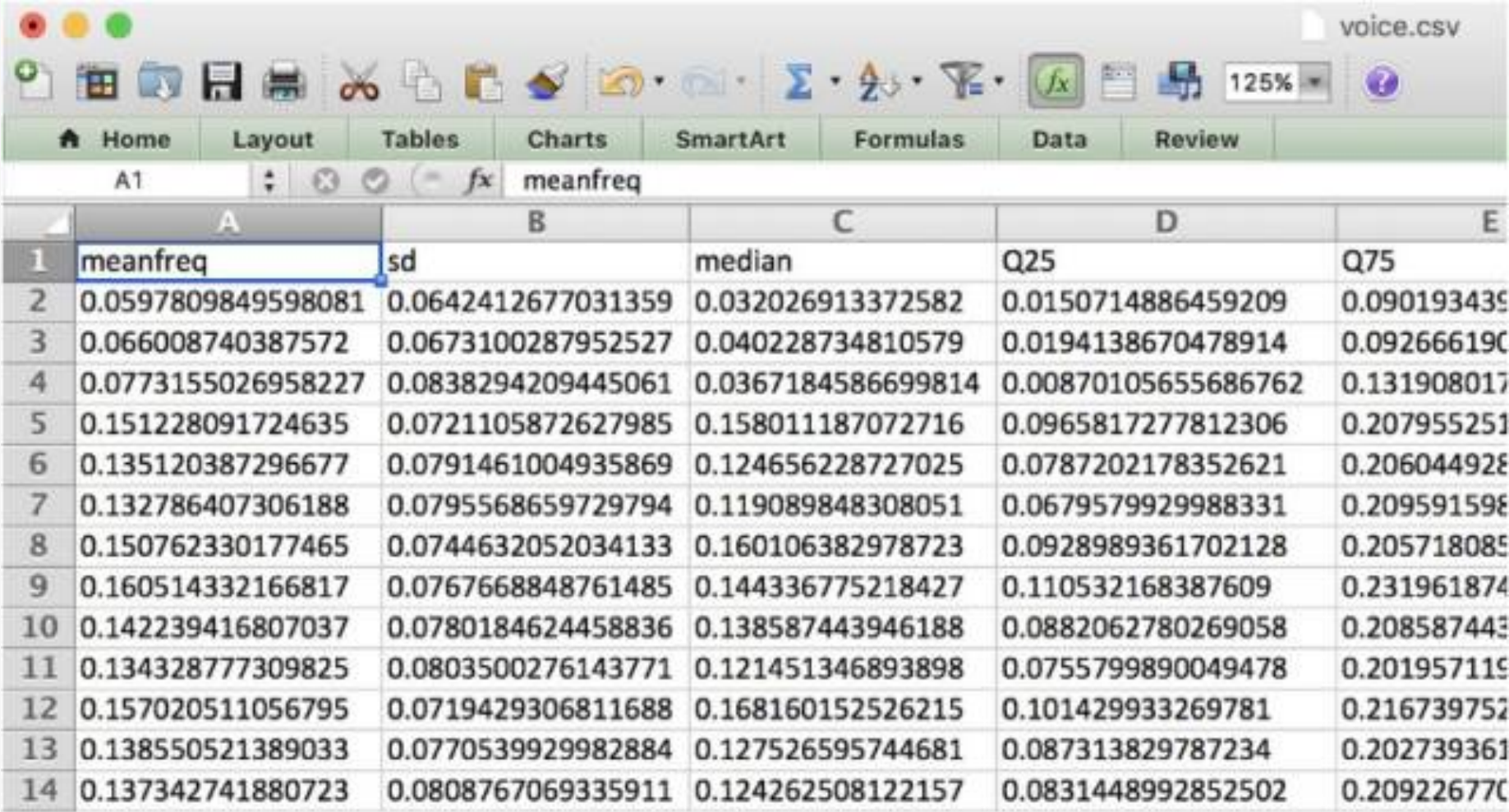
## 仔细观察数据

要训练一个分类器，你需要数据。

对于这个项目，我们使用 Kory Becker 的「Gender Recognition by Voice」数据集，这是我在 Kaggle 网站上找到的。

下载地址：<https://www.kaggle.com/primaryobjects/voicegender>

那么你如何从音频里识别语音？如果你下载了该数据集并查看 voice.csv 文件，你将只会看到一行又一行的数字：



	A	B	C	D	E
1	meanfreq	sd	median	Q25	Q75
2	0.0597809849598081	0.0642412677031359	0.032026913372582	0.0150714886459209	0.090193435
3	0.066008740387572	0.0673100287952527	0.040228734810579	0.0194138670478914	0.092666190
4	0.0773155026958227	0.0838294209445061	0.0367184586699814	0.00870105655686762	0.131908017
5	0.151228091724635	0.0721105872627985	0.158011187072716	0.0965817277812306	0.207955251
6	0.135120387296677	0.0791461004935869	0.124656228727025	0.0787202178352621	0.206044928
7	0.132786407306188	0.0795568659729794	0.119089848308051	0.0679579929988331	0.209591598
8	0.150762330177465	0.0744632052034133	0.160106382978723	0.0928989361702128	0.205718085
9	0.160514332166817	0.0767668848761485	0.144336775218427	0.110532168387609	0.231961874
10	0.142239416807037	0.0780184624458836	0.138587443946188	0.0882062780269058	0.208587443
11	0.134328777309825	0.0803500276143771	0.121451346893898	0.0755799890049478	0.201957119
12	0.157020511056795	0.0719429306811688	0.168160152526215	0.101429933269781	0.216739752
13	0.138550521389033	0.0770539929982884	0.127526595744681	0.087313829787234	0.202739361
14	0.137342741880723	0.0808767069335911	0.124262508122157	0.0831448992852502	0.209226770

要意识到这不是实际的音频数据，这是很重要的！相反，这些数字表示的是音频的不同声学特性。这些属性（或特征）是通过脚本从音频中提取出来并转换为此 CSV 文件的。对这个过程的解释超出了本教程的范围，但如果你有兴趣，可以参考原始的 R 语言源代码：<https://github.com/primaryobjects/voice-gender>

该数据集包含 3,168 个这样的样本（每个对应表格中的每一行），大约一半是男性说话者，一半是女性说话者。其中每个样本有 20 个声学特征，比如：

平均频率（kHz）



频率的标准差

音谱平坦度

音谱熵

峰度

声信号测量的最大基频

调制指数

等等...

对于大多数这些，我不知道它们的意思，而且这也不是很重要。我们所关心的是，我们可以使用这些数据来训练分类器，以便根据这些特征来分辨男性和女性的声音。

如果你想在应用程序中使用此分类器来分辨来自麦克风的音频或录音中说话人的性别，那么你必须首先从音频数据中提取这些声学属性。一旦你有这 20 个数值，你可以把它们提供给训练好的分类器，而它会告诉你这个声音是男性还是女性。所以我们的分类器不会直接在录音上工作，而只是在从录音中提取的特征上工作。

注：此处可以很好地指出深度学习和更传统的算法（如 logistic 回归）之间的区别。我们正在训练的分类器不能学习非常复杂的事情，你需要通过在预处理步骤中从数据里提取特征来帮助它。对于这个特定数据集来说，就是从音频中提取声学特征。

关于深度学习的很酷的地方是，你可以训练神经网络来学习如何让它自己来提取这些声学特征。因此，深度学习系统可以将原始音频作为输入，提取其认为重要的声学特征，然后进行分类，而无需进行任何预处理。

## 创建训练集和测试集

前面我提到我们通过以下方式训练分类器：

1. 提供给它数据集中所有的样本
2. 测量预测的错误程度
3. 根据损失调整权重

实际上我们不应该使用所有的数据进行训练。我们需要将一部分数据（称为测试集）分离出来，以便我们可以评估我们的分类器的效果。因此，我们将数据集分为两部分：我们用于训练分类器的训练集，以及我们用来查看分类器的准确度的测试集。

为了将数据分成训练集和测试集，我创建了一个名为 `split_data.py` 的 Python 脚本：

```
import numpy as np          # 1
import pandas as pd

df = pd.read_csv("voice.csv", header=0)    # 2

labels = (df["label"] == "male").values * 1 # 3
labels = labels.reshape(-1, 1)             # 4

del df["label"]                    # 5
data = df.values

# 6
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data, labels,
                                                    test_size=0.3, random_state=123456)

np.save("X_train.npy", X_train) # 7
np.save("X_test.npy", X_test)
np.save("y_train.npy", y_train)
np.save("y_test.npy", y_test)
```

一步一步来讲，这个脚本是这样工作的：

导入 NumPy 和 pandas 包。pandas 可以让我们可以轻松加载 CSV 文件，并对数据进行预处理。

使用 pandas 将数据集从 `voice.csv` 加载到所谓的 `dataframe` 中。此对象的工作原理非常像电子表格或 SQL 表。

`label` 列包含该数据集的标签：样本是男还是女。这里我们将标签提取到一个新的 NumPy 数组中。原始标签是文本，但我们将其转换为数字：1=男性，0=女性。（这些数字的分配是任意的——在二值分类器中，我们经常使用 1 来表示「正」类，或者说我们试图检测到的类）。

这个新的 `labels` 数组是一维数组，但是我们的 TensorFlow 脚本将会有有一个 3,168 行的二维张量，其每行有一列。所以在这里我们将该数组「重塑」成二维。这不会改变内存中的数

据，只是改变从现在起 NumPy 解读这些数据的方式。

一旦我们完成了 label 列，我们将其从 dataframe 中删除，这样我们便留下了用来描述该输入的 20 个特征。我们也将该 dataframe 转换为了一个常规的 NumPy 数组。

我们使用 scikit-learn 的一个辅助函数将 data 和 labels 数组拆分成两部分。这随机地将基于 random\_state 的数据集中的样本进行重排，random\_state 是随机生成器的种子。这个种子是什么无关紧要，但如果总是使用相同的种子，我们就可以创建一个可复现的实验。

最后，以 NumPy 的二进制文件格式保存四个新的数组。我们现在就有了一个训练集和一个测试集！

你可以对此脚本中的数据进行其它预处理（例如扩展特征），使其具有零均值和相同的方差；但是我没将其放在这个简单的项目中。

从终端运行如下脚本：

这给了我们四个包含训练样本（X\_train.npy）及其标签（y\_train.npy）和测试样本（X\_test.npy）及其标签（y\_test.npy）的新文件。

注：你可能会想知道为什么一些变量名称被大写，为什么别的没有。在数学中，矩阵通常被写成大写字母而向量为小写。在我们的脚本中，X 是矩阵，y 是向量。这样的惯例在大量机器学习代码中很常见。

## 构建计算图

现在我们将数据整理好了，我们可以编写一个用 TensorFlow 训练 logistic 分类器的脚本。这个脚本叫做 train.py。为了节省空间，我不会在这里展示整个脚本，你可以在 GitHub 上看到它。

同样，我们首先导入我们需要的包。然后我们将训练数据加载到两个 NumPy 数组中：X\_train 和 y\_train。（我们不会在这个脚本中使用测试数据。）

```
import numpy as np
import tensorflow as tf

X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")
```

现在我们可以创建我们的计算图。首先我们为我们的输入  $x$  和  $y$  定义所谓的占位符 (placeholder)：

```
num_inputs = 20
num_classes = 1

with tf.name_scope("inputs"):
    x = tf.placeholder(tf.float32, [None, num_inputs], name="x-input")
    y = tf.placeholder(tf.float32, [None, num_classes], name="y-input")
```

`tf.name_scope`（「...」）可用于将图的不同部分分组到不同的范围 (scope)，这样可以更容易地了解图。我们把  $x$  和  $y$  放在「input」范围内。我们还给它们命名「x-input」和「y-input」，这样我们稍后可以很容易地引用它们。

回想一下，每个输入样本就是一个 20 个元素的向量。每个样本也有一个标签（1 是男，0 是女）。我还提到，如果我们将所有的样本合并成一个矩阵，我们可以一次性计算所有的数据。这就是为什么  $x$  和  $y$  在这里被定义为二维张量： $x$  的维度是 `[None, 20]`， $y$  的维度是 `[None, 1]`。

`None` 意味着第一维度是灵活的而且尚不知道。对于训练集，我们将把 2,217 个样本放在  $x$  和  $y$  中；对于测试集，则是 951 个样本。

现在，TensorFlow 知道我们的输入是什么，我们可以定义分类器的参数 (parameter)：

```
with tf.name_scope("model"):
    W = tf.Variable(tf.zeros([num_inputs, num_classes]), name="W")
    b = tf.Variable(tf.zeros([num_classes]), name="b")
```

张量  $W$  包含分类器将要学习的权重（一个  $20 \times 1$  矩阵，因为有 20 个输入特征和 1 个输出）， $b$  包含偏置值。这两个被声明为 TensorFlow 变量，这意味着它们可以通过反向传播过程进行更新。

就绪之后，我们可以声明在我们的 logistic 回归分类器的核心位置的计算：

其中  $x$  和  $W$  相乘，加上偏差  $b$ ，然后执行 logistic sigmoid。 $y_{\text{pred}}$  的结果是由  $x$  中的特征所描述的音频数据的说话人是男性的概率。

注：上面的代码行实际上并没有计算任何东西——所有我们一直在做的是创建计算图。这行代码是简单地将节点添加到图中用于矩阵乘法 (`tf.matmul`)、加法 (+) 和 Sigmoid 函数



(tf.sigmoid)。一旦我们构建了整个图，我们可以创建一个 TensorFlow 会话并在实际数据上运行它。

我们还没有完成。为了训练该模型，我们需要定义一个损失函数（loss function）。对于二值 logistic 回归分类器，使用损失日志 log loss 是有意义的，幸运的是 TensorFlow 有一个内置的 log\_loss() 函数，可以避免我们写出实际的数学公式：

```
with tf.name_scope("loss-function"):
    loss = tf.losses.log_loss(labels=y, predictions=y_pred)
    loss += regularization * tf.nn.l2_loss(w)
```

log\_loss 图节点以 y（我们当前正在查看的样本的标签）作为输入，并将它们与我们的预测 y\_pred 进行比较。这就产生了一个代表损失的数字。

当我们开始训练时，对于所有的样本，预测 y\_pred 将为 0.5（即 50% 的概率是男性），因为分类器不知道正确的答案应该是什么。因此，以  $-\ln(0.5)$  计算的初始损失为 0.693146。随着训练的进行，损失将越来越小。

计算损失的第二行增加了一些称为 L2 正则化（L2 regularization）的东西。这样做是为了防止过拟合，使得分类器无法确切地记忆训练数据。在这里，这不会是一个很大的问题，因为我们的分类器的「记忆」只包含 20 个权重值和一个偏置值。但是正则化是一种常见的机器学习技术，所以我以为我会包括它。

regularization 值是另一个占位符：

```
with tf.name_scope("loss-function"):
    loss = tf.losses.log_loss(labels=y, predictions=y_pred)
    loss += regularization * tf.nn.l2_loss(w)
```

我们已经使用了占位符来定义我们的输入 x 和 y，但它们也可用于定义超参数（hyperparameter）。超参数可让你配置模型和训练方式。它们被称为「超」参数，因为与常规参数 W 和 b 不同，它们不被模型学习——你必须自己将它们设置为适当的值。

learning\_rate（学习率）超参数告诉优化器应该采取多大的步伐。优化器（optimizer）是执行反向传播的：它以损失为输入，并将其反馈到图中，以确定更新权重和偏置的程度。有很多可能的优化器，我们将使用 ADAM：

```
with tf.name_scope("hyperparameters"):
    regularization = tf.placeholder(tf.float32, name="regularization")
    learning_rate = tf.placeholder(tf.float32, name="learning-rate")
```

这将在图中创建一个名为 train\_op 的节点。这是我们稍后将运行的节点，以便训练分类器。

为了确定分类器的运行情况，我们将在训练期间偶尔进行快照，并记数出训练集中已经正确预测的样本个数。训练集上的准确性并不是分类器工作的美好指标，但是无论如何，它对跟踪训练是有用的——如果你正在进行训练，并且训练集上的预测准确性变得更糟，那么一定是哪里出现了问题！

我们定义一个图节点来计算准确性：

```
with tf.name_scope("score"):
    correct_prediction = tf.equal(tf.to_float(y_pred > 0.5), y)
    accuracy = tf.reduce_mean(tf.to_float(correct_prediction), name="accuracy")
```

我们可以运行 accuracy 节点来查看有多少样本被正确预测。回想一下，y\_pred 包含 0 到 1 之间的概率。通过做 tf.to\_float (y\_pred> 0.5)，如果预测是女性，我们得到一个值 0，如果预测是男性，则得到 1。我们可以将其与 y 进行比较，y 中包含正确的值。那么准确性就是正确的预测数除以预测的总数。

之后，我们还将在测试集上使用同一个 accuracy 节点，以了解分类器的真正效果。

多定义一个节点很有用。这一个节点用于对我们根本没有任何标签的数据进行预测：

要在应用程序中使用此分类器，你要录制一些话，分析它以提取 20 个声学特征，然后将其提供给分类器。因为这是新的数据，不是来自训练集或测试集的数据，显然不会有标签。你只能将此新数据提供给分类器，并希望它预测正确的结果。这就是 inference（推理）节点所需要做的。

好的，之前那么多工作仅仅就是为了创建计算图。现在我们想在训练集上实际训练它。

## 训练分类器

训练通常是一个无限循环的过程。要训练一个简单的、功能稍微强大的 logistic 回归分类器，一般一分钟之内就能完成，但是如果要训练一个性能优异的深度神经网络，可能需要花费几个小时甚至几天时间才能完成。

以下是「train.py」文件中训练回路的第一部分：

```
with tf.Session() as sess:
    tf.train.write_graph(sess.graph_def, checkpoint_dir, "graph.pb", False)

    sess.run(init)

    step = 0
    while True:
        # here comes the training code
```

首先，我们在 TensorFlow 中创建一个新的 session 对象。为了运行计算图，你需要首先启动会话（session）。调用 sess.run(init) 将 W 和 b 重置为 0。

我们也将此计算图写入到了一个文件。将我们刚才创建的所有节点序列化到文件/tmp/voice/graph.pb 中，稍后在测试集上运行分类器时，我们需要这个定义图，我们也可以将这个训练好的分类器放入 iOS 应用程序中。

在这个 while True 中，我们执行以下命令：

```
with tf.Session() as sess:
    tf.train.write_graph(sess.graph_def, checkpoint_dir, "graph.pb", False)

    sess.run(init)

    step = 0
    while True:
        # here comes the training code
```

首先，我们将训练样本随机混洗——这一步很重要，因为你不希望这个分类器无意中学习到与样本顺序有关的信息。

接下来这一点很重要：我们告诉 session 去运行 train\_op 结点，这将在计算图上执行单次训练。

当你运行 sess.run() 时，你需要提供一个 feed\_dict，feed\_dict 是告诉你想占位符（placeholder）结点中相应的输入数据。

由于这是一个非常简单的分类器，因此我们总是一次性地训练整个训练集。我们将 X\_train, y\_train 的训练数据分别传入占位符 x, y 中。对于较大的数据集，你可以采用分批训练的方法，比如以每批 100-1000 个样例进行训练。

这些都是我们需要做的。训练是一个循环过程，因此 train\_op 结点要运行很多很多次。在每一次迭代过程中，反向传播机制就会使权重 W 和 b 做出微小的变化。多次训练后，我们一



般能得到权重的最优或较优值。

理解训练的过程对于我们理解神经网络是非常有帮助的，所以我们将训练过程的进度报告打印出来（本项目中，每 1000 步打印一次）：

```
if step % print_every == 0:
    train_accuracy, loss_value = sess.run([accuracy, loss],
                                          feed_dict=feed)
    print("step: %4d, loss: %.4f, training accuracy: %.4f" % \
          (step, loss_value, train_accuracy))
```

这一次，我们不是运行 train\_op 结点，而是其它结点：准确率（accuracy）和损失函数（loss）。我们使用了相同的 feed\_dict，以便于在训练集上计算准确率和损失函数。

正如我之前所说，在训练集上表现高准确度的分类器并不一定在测试集上表现也很好。但是有一点很肯定，那就是你很希望看到准确率这个指标随着训练而不断上升，损失函数值不断减少。

我们很多时候都会保存一个检查点（checkpoint）。

```
if step % save_every == 0:
    checkpoint_file = os.path.join(checkpoint_dir, "model")
    saver.save(sess, checkpoint_file)
    print("*** SAVED MODEL ***")
```

将分类器学习到的 W 和 b 的值保存到一个 checkpoint 文件中，当我们想在测试集上运行该分类器的时候，我们将再次读取 checkpoint 文件中的数据。checkpoint 文件保存在/tmp/voice/目录下。

在终端输入如下命令运行训练脚本：

输出应该是像这样的：



```
Training set size: (2217, 20)
Initial loss: 0.693146
step: 0, loss: 0.7432, training accuracy: 0.4754
step: 1000, loss: 0.4160, training accuracy: 0.8904
step: 2000, loss: 0.3259, training accuracy: 0.9170
step: 3000, loss: 0.2750, training accuracy: 0.9229
step: 4000, loss: 0.2408, training accuracy: 0.9337
step: 5000, loss: 0.2152, training accuracy: 0.9405
step: 6000, loss: 0.1957, training accuracy: 0.9553
step: 7000, loss: 0.1819, training accuracy: 0.9594
step: 8000, loss: 0.1717, training accuracy: 0.9635
step: 9000, loss: 0.1652, training accuracy: 0.9666
*** SAVED MODEL ***
step: 10000, loss: 0.1611, training accuracy: 0.9702
step: 11000, loss: 0.1589, training accuracy: 0.9707
. . .
```

一旦你看到损失函数停止下降时，保持等待，直到你看见\*\*\* SAVED MODEL \*\*\*的提示信息，然后在计算机上按下 Ctrl+C 停止训练。使用我选择的正则化参数和学习率，你应该看到在训练集上的准确率大约为 97%，损失函数约为 0.157（如果你将正则化参数设置为 0，损失函数值将更小）。

## 分类器的表现如何？

训练好分类器后，我们需要测试它在实际生活中的表现如何。那么你就需要使用没有用于训练的数据来评估分类器，这就是为什么我们将数据集分为训练集合测试集。

我们创建了一个新的脚本 test.py，用于加载定义好的计算图和测试集，最终计算出在测试集中的分类准确率。

注：测试准确率总是低于训练准确率（本文为 97%），但是也不会低太多。如果你的分类器在测试集上准确率远不及在训练集上准确率，你的分类器很可能存在过拟合，这时你需要重新调整你的训练程序。我们期望此分类器在测试集上的准确率达 95%，如果低于 90%，那么就需要引起你的注意了。

像之前一样，脚本中首先导入相关的包。包括使用 scikit-learn 中的 metrics 包来打印一些相关报告。当然，这次我们加载的是测试集而不是训练集。

```
import numpy as np
import tensorflow as tf
from sklearn import metrics

X_test = np.load("X_test.npy")
y_test = np.load("y_test.npy")
```

为了计算测试集上的准确率，我们需要再次使用我们的计算图，但这并不是使用整个计算图，此处我们不需要使用 `train_op` 和 `loss` 结点。

我们可以再次手动构建此图，但是由于我们已经保存在了 `graph.pb` 文件中，我们只需要加载它而已。代码如下：

```
with tf.Session() as sess:
    graph_file = os.path.join(checkpoint_dir, "graph.pb")
    with tf.gfile.GFile(graph_file, "rb") as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        tf.import_graph_def(graph_def, name="")
```

TensorFlow 喜欢将其数据存储为协议缓冲区文件（扩展名为.pb），因此我们使用一些帮助代码来加载该文件，并将其作为图形导入到会话中。

接下来，我们需要从 `checkpoint` 文件中加载训练好的 `W` 和 `b` 值：

```
W = sess.graph.get_tensor_by_name("model/W:0")
b = sess.graph.get_tensor_by_name("model/b:0")

checkpoint_file = os.path.join(checkpoint_dir, "model")
saver = tf.train.Saver([W, b])
saver.restore(sess, checkpoint_file)
```

这就是为什么我们将结点限制范围并给它们相应的名字，所以我们只需要使用 `get_tensor_by_name()` 函数就能够很方便地再次找到它们。如果你没有为你的结点取上意义明确的名字。那么你要找到它们就麻烦了。

我们也需要获得一些结点的引用（references），特别是输入 `x`，`y` 以及进行预测的结点。

```
x = sess.graph.get_tensor_by_name("inputs/x-input:0")
y = sess.graph.get_tensor_by_name("inputs/y-input:0")
accuracy = sess.graph.get_tensor_by_name("score/accuracy:0")
inference = sess.graph.get_tensor_by_name("inference/inference:0")
```

OK，目前为止，我们已经将计算图加载到内存中。我们也已经加载好了先前分类器训练好的



W 和 b。现在我们可以测试集（以前未见过的数据集）中测试。

使用 X\_test 做预测，将预测值与标签 y\_test 做对比，验证预测是否准确并计算准确率。

注意：这次在 feed\_dict 中不需要指定正则化参数和学习速率。我们仅需要运行计算图中的相关部分，不需要使用计算图中的占位符部分。我们也可以利用 scikit-learn 包来生产一些其它报告：

```
predictions = sess.run(inference, feed_dict={x: X_test})
print("Classification report:")
print(metrics.classification_report(y_test.ravel(), predictions))
print("Confusion matrix:")
print(metrics.confusion_matrix(y_test.ravel(), predictions))
```

这次我们使用 inference 结点做预测。由于 inference 仅仅计算预测值而不检查预测值的准确率，因此只需要向 feed\_dict 传入 x（不需要 y）。

运行此脚本，您应该看到类似如下的输出：

```
$ python3 test.py
```

```
Test set accuracy: 0.958991
```

```
Classification report:
```

	precision	recall	f1-score	support
0	0.98	0.94	0.96	474
1	0.94	0.98	0.96	477
avg / total	0.96	0.96	0.96	951

```
Confusion matrix:
```

```
[[446  28]
 [ 11 466]]
```

在测试集上的准确率几乎达到了 96%，正如预期所说，测试集上的准确率要低于训练集上的

准确率。这意味着我们的训练是相当成功的，我们的模型在未知数据上表现也很成功。这还不够完美：在每 25 次预测中几乎会犯错一次。但是对于我们的目的而言，这已经很好了。

分类报告和混淆矩阵展示了被错误预测样例的统计数据。从混淆矩阵来看，有 446 名女性被正确预测，有 28 名女性被误认为男性；466 名男性被正确预测，11 名男性被误认为女性。

这似乎该分类器在预测女性过程中更容易犯错，来自分类报告的准确率/召回率也说明了相同的事实。

## 在 iOS 上构建 TensorFlow

现在，我们已经训练了一个在测试集上表现很好的模型，让我们建立一个简单的利用该模型做预测的 iOS 应用程序。首先，我们将制作一个利用 TensorFlow C++ 库的应用程序。在下一节中，我们将此模型用于 Metal 中作比较。

当然，这样做既有好处也有坏处。坏消息是你必须从源构建 TensorFlow。实际上，它会变得更糟：你需要安装 Java 来实现。好消息是这个过程相对较简单。完整的指导说明你能在这里（Full instructions are here ([https://www.tensorflow.org/install/install\\_sources](https://www.tensorflow.org/install/install_sources))）找到，但是还需要以下几个步骤（在 TensorFlow 1.0 上做测试）。

当然，你应该安装好了 Xcode，并确保激活指向 Xcode 安装的开发者的目录（如果你在安装 Xcode 之前已经安装好了 Homebrew，这可能会指向错误的位置，这种情况下，TensorFlow 将不会安装成功。）

建立 TensorFlow 需要使用一款叫做 bazel 的工具，bazel 需要 Java JDK 8 支持。使用 Homebrew 很容易安装你所需要的包：



```
brew cask install java  
brew install bazel  
brew install automake  
brew install libtool
```

一旦你完成上述步骤，你需要克隆 TensorFlow 的 GitHub 仓库。注意：将此保存在没有空格的路径中，否则 bazel 将拒绝构建 TensorFlow！我将此 GitHub 仓库简单地克隆到我的主目录下：

-b r1.0 标识符表示克隆 r1.0 分支。当然，你也可以随时自由获取最新的分支或主要分支。

注意：在 macOS Sierra 上，运行下面的配置脚本将会给出一些错误。我不得不以克隆主分支来代替。在 OS X El Capitan 上，r1.0 分支没有错误。

完成克隆步骤后，你需要运行配置脚本。

```
cd tensorflow  
./configure
```

这里你可能会遇到一些的问题，我在此给出它们的答案：

请指定 python 的位置，默认路径是/usr/bin/python。因为我想使用 Python3.6 版本，因此路径应为 /usr/local/bin/python3。如果你选择默认选项，TensorFlow 将建立在 Python2.7 之上。

请指定编译期间需要使用的优化标识符，默认是-march=native。

只需按下回车确认即可。接下来的几个问题选择「n」代替「no」即可。

当他询问要使用哪个 Python 库的时候，请按下 Enter 选择默认选项（这应该是 Python3.6 库）。

剩下的几个问题用「n」代替「no」来回答即可。现在脚本将下载一些依赖并为建立 TensorFlow 做准备。

## 建立静态库

构建 TensorFlow 有如下两个选项：

1. 在 Mac 上，使用 bazel 工具
2. 在 iOS 上，使用 Makefile

我们想为 iOS 构建 TensorFlow，所以我们使用第二个选项。然而，我们也将需要构建一些与选项一有关的额外工具。

在 TensorFlow 的目录下中执行以下脚本：

首先下载几个依赖，然后启动构建程序。如果一切都顺利的话，它将创建三个需要链接到你应用程序的静态库：libtensorflow-core.a，libprotobuf.a，libprotobuf-lite.a。

警告：建立这些库需要花费一段时间：在我的 iMac 上，用了将近 20 多分钟，在相对较老的 MacBook Pro 上，则花费了超过 3 个小时。在安装过程中，你也可能会看到很多编译警告信息，甚至错误信息。最简单的处理方式：先忽略它们。

现在，我们还需要另外另个辅助安装工具。在终端运行下面两条命令：

注意：这将需要花费 20 分钟左右的时间，因为它是从头开始构建 TensorFlow（此次使用 bazel）；如果此过程中遇到麻烦，请参阅官方说明。

## 为 Mac 构建 TensorFlow

这一步是可选项，但是由于你已经安装好了所有依赖环境，所以要为你的 Mac 建立 TensorFlow 一点儿也不困难。这创建了一个新的 pip 包，因此你不需要利用官方 TensorFlow 包进行安装。

为什么要这样做呢？因为这样你就可以创建一个具有自定义选项的 TensorFlow 版本。例如，当你在运行 train.py 文件时，如果你得到「The TensorFlow library wasn't compiled to use SSE4.1 instructions」的警告信息时，你可以编译一个能执行这些指令的 TensorFlow 版本。

要为 Mac 构建 TensorFlow，在终端运行运行下面的命令：

```
bazel build --copt=-march=native -c opt //tensorflow/tools/pip_package:build_pip_package
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

「-march=native option」选项对 SSE, AVX, AVX2, FMA 等指令（如果这些指令能够在你的 CPU 上可用的话）增加了支持。

然后安装下面的包：

更多详细说明，请参考 TensorFlow 官网。

## Freezing the graph

我们将建立的 iOS 应用程序将加载我们曾经训练好的模型，并用此应用程序来做一些预测。

回忆一下，train.py 文件将计算图的定义保存到了/tmp/voice/graph.pb 之中。很遗憾的是，你不能将此图原样地加载到 iOS 应用程序中。完整的计算图包含某些不受 TensorFlow C++ API 的支持的操作。这就是为什么我们需要使用两个额外工具的原因。

freeze\_graph 能将 graph.pb 文件和保存模型训练好的超参数值 W 和 b 的 checkpoint 文件合并到一个文件中，它还能够移除 iOS 不支持的任何操作。

在终端的 TensorFlow 的目录下运行此工具：

```
bazel-bin/tensorflow/python/tools/freeze_graph \
--input_graph=/tmp/voice/graph.pb --input_checkpoint=/tmp/voice/model \
--output_node_names=model/y_pred,inference/inference --input_binary \
--output_graph=/tmp/voice/frozen.pb
```

这就在 /tmp/voice/frozen.pb 文件中创建了一个简化图，其中只包含到 y\_pred 和 inference 的所有结点，它不包括用于训练的结点。

使用 freeze\_graph 的优势在于它能够将训练好的权重粘贴到文件中，所以你不需要单独加载它们： frozen.pb 包含了我们所需要的一切内容。

optimize\_for\_inference 工具将进一步简化图，它将 frozen.pb 文件作为输入，写入 /tmp/voice/inference.pb 文件作为输出。这个输出文件就是我们将要嵌入在 iOS 应用程序中的文件，使用如下命令运行此工具。

```
bazel-bin/tensorflow/python/tools/optimize_for_inference \  
--input=/tmp/voice/frozen.pb --output=/tmp/voice/inference.pb \  
--input_names=inputs/x --output_names=model/y_pred,inference/inference \  
--frozen_graph=True
```

## iOS 应用程序

你可以在 VoiceTensorFlow 文件夹中的 [github.com/hollance/TensorFlow-iOS-Example](https://github.com/hollance/TensorFlow-iOS-Example) 中找到该 iOS 应用程序。

在 Xcode 中打开项目，你需要注意如下几点：

该应用程序是用面向对象的 C++ 语言写成的，源文件后缀为 .mm。这里没有用到 TensorFlow 的 Swift API，只用到了 C++。

inference.pb 已经存在于该项目中，你可以将自己的 inference.pb 版本复制到项目文件夹中。

此应用程序连接到 Accelerate.framework 上。

此应用程序与你编译的静态库链接。

转到项目设置屏幕并切换到构建设置选项卡。在其他链接器标识符下，你将看到以下内容：



```
/Users/matthijs/tensorflow/tensorflow/contrib/makefile/gen/protobuf_ios/lib/  
libprotobuf-lite.a  
  
/Users/matthijs/tensorflow/tensorflow/contrib/makefile/gen/protobuf_ios/lib/  
libprotobuf.a  
  
-force_load /Users/matthijs/tensorflow/tensorflow/contrib/makefile/gen/lib/  
libtensorflow-core.a
```

除非你的命名也是「matthijs」，否则你将需要用克隆的 TensorFlow 仓库路径来代替它。  
(注意 TensorFlow 出现两次，因此文件夹名称应该为 tensorflow / tensorflow / ...)

注意：你也可以将这三个.a 文件复制到项目文件夹中，然后你就不需要担心路径出错的问题了。对于这个项目我不想这样做，因为 libtensorflow-core.a 太大，是一个占用 440MB 的库。

然后检查标题搜索路径 (\*\*Header Search Paths\*\*)，它们当前位置是：

```
~/tensorflow  
~/tensorflow/tensorflow/contrib/makefile/downloads  
~/tensorflow/tensorflow/contrib/makefile/downloads/eigen  
~/tensorflow/tensorflow/contrib/makefile/downloads/protobuf/src  
~/tensorflow/tensorflow/contrib/makefile/gen/proto
```

最后，你必须将这些路径更新到克隆仓库的位置。

这里还有一些设置，如下：

Enable Bitcode: No

Warnings / Documentation Comments: No

Warnings / Deprecated Functions: No

TensorFlow 当前不支持 Bitcode，所以我把它禁用了。我也关闭了警告，否则在你编译应用程序时，你会遇到很多问题（你仍然将会收到几个 Value Conversion Issue 警告，当然你也可以禁用这些警告，但我很喜欢这些警告信息）。

一旦你对其它链接器标志 (Other Linker Flags) 和标题搜索路径 (Header Search Paths) 做出更改后，你就可以建立并运行这个应用程序了。

很好，现在你已经有一个使用了 TensorFlow 的 iOS 应用程序！让我看看它是怎样工作的。

## 使用 TensorFlow C++ API

iOS 上的 TensorFlow 是用 C++编写的，但是你需要编写的 C++代码的程序是非常有限的！这一点你很幸运。通常你将执行以下一些操作：

1. 加载.pb 文件中的权重和图；
2. 使用图时先启动会话；
3. 将你的输入数据放入输入张量；
4. 在一个或多个结点上运行计算图；
- 5 得到输出结点的输出张量值，

在这个演示的应用程序中，这一切都在 ViewController.mm 文件中完成。首先，让我们加载计算图：

```
- (BOOL)loadGraphFromPath:(NSString *)path
{
    auto status = ReadBinaryProto(tensorflow::Env::Default(),
                                   path.fileSystemRepresentation, &graph);
    if (!status.ok()) {
        NSLog(@"Error reading graph: %s", status.error_message().c_str());
        return NO;
    }
    return YES;
}
```

Xcode 项目包括了 inference.pb 图，它是由我们在 graph.pb 上运行 freeze\_graph 和 optimize\_for\_inference 得到的。如果你试图加载 graph.pb，你将得到一些错误信息：

```
Error adding graph to session: No OpKernel was registered to support Op
'L2Loss' with these attrs. Registered devices: [CPU], Registered kernels:
<no registered kernels>

[[Node: loss-function/L2Loss = L2Loss[T=DT_FLOAT](model/W/read)]]
```

该 C++ API 支持的操作比 Python API 支持的操作少。在这里，在我们损失函数结点的

L2Loss 操作在 iOS 上是不可用的。这就是为什么我们使用 freeze\_graph 来简化我们的图的原因。在加载图后，我们开始启动会话：

```
- (BOOL)createSession
{
    tensorflow::SessionOptions options;
    auto status = tensorflow::NewSession(options, &session);
    if (!status.ok()) {
        NSLog(@"Error creating session: %s",
              status.error_message().c_str());
        return NO;
    }

    status = session->Create(graph);
    if (!status.ok()) {
        NSLog(@"Error adding graph to session: %s",
              status.error_message().c_str());
        return NO;
    }
    return YES;
}
```

一旦我们启动了会话，我们就能够利用它做一些预测。预测：将包含 20 个浮点数的数组作为声学特征传入计算图。

让我们看看该方法是怎样工作的：

```
- (void)predict:(float *)example {
    tensorflow::Tensor x(tensorflow::DT_FLOAT,
                        tensorflow::TensorShape({ 1, 20 }));

    auto input = x.tensor<float, 2>();
    for (int i = 0; i < 20; ++i) {
        input(0, i) = example[i];
    }
}
```

首先，我们定义输入数据张量 x，该张量的形状为 ，即 1 个样例，20 个特征。然后将我们



的数据从数组转换成 TensorFlow 中的张量。

接下来，我们运行会话：

```
- (void)predict:(float *)example {
    tensorflow::Tensor x(tensorflow::DT_FLOAT,
                          tensorflow::TensorShape({ 1, 20 }));

    auto input = x.tensor<float, 2>();
    for (int i = 0; i < 20; ++i) {
        input(0, i) = example[i];
    }
}
```

使用如下类似 Python 中的代码，看看发生了什么：

这条命令并不那么简洁，我们创建了 feed\_dict，结点矢量，以及保存结果的一个矢量。最后，我们告诉会话来做我们想做的事情。

一旦启动会话，运行了结点，我们就能打印出结果：

```
auto y_pred = outputs[0].tensor<float, 2>();
NSLog(@"Probability spoken by a male: %f%%", y_pred(0, 0));

auto isMale = outputs[1].tensor<float, 2>();
if (isMale(0, 0)) {
    NSLog(@"Prediction: male");
} else {
    NSLog(@"Prediction: female");
}
}
```

出于我们的目的，仅仅运行到 inference 结点就足够了，但是我也想看看预测的准确率，所以我们也运行了 y\_pred 结点。

## 运行应用程序

你可以在 iPhone 模拟器或其它设备上运行该应用程序。在模拟器上，你可能会再次接到「The TensorFlow library wasn't compiled to use SSE4.1 instructions」的消息，但是在设备上，你不应该会接收到这些消息。



仅仅用于测试的目的，该应用程序将仅仅做出两类预测：预测男性或女性。我们可以仅仅从测试集中随机取出相应的数据来做预测。

运行该应用程序，你应该看到以下输出。该应用程序首先打印出图中的节点：

```
Node count: 9
Node 0: Placeholder 'inputs/x-input'
Node 1: Const 'model/W'
Node 2: Const 'model/b'
Node 3: MatMul 'model/MatMul'
Node 4: Add 'model/add'
Node 5: Sigmoid 'model/y_pred'
Node 6: Const 'inference/Greater/y'
Node 7: Greater 'inference/Greater'
Node 8: Cast 'inference/inference'
```

注意：此图仅仅包括了进行预测所需的操作，并没有给出训练信息。然后打印预测结果：

```
Probability spoken by a male: 0.970405%
Prediction: male

Probability spoken by a male: 0.005632%
Prediction: female
```

如果你在 Python 脚本中尝试相同的样例，你将得到完全相同的答案。我们的任务终于完成了！

注意：对于这个演示项目，我们使用的数据仅仅是从测试集中抽取出来。如果你想使用该模

型去处理实际音频，那么你需要将音频转换成 20 种声学特征。

## iOS 行 TensorFlow 的优点和缺点

TensorFlow 是一款强大的用于训练机器学习模型和实现新算法的框架。为了训练大模型，你甚至可以在云端使用 TensorFlow。

本文除了讲述如何训练模型外，还展示了如何将 TensorFlow 添加到你的 iOS 应用程序中。在本节中，我想总结一下这样做的优点与缺点。

iOS 上使用 TensorFlow 的优点：

使用一个工具做所有事情。一方面，你可以使用 TensorFlow 训练模型，也可以进行推理，这不需要将你的计算图从 TensorFlow 移植到其他的 API，例如 BNNS 或 Metal 上；另一方面，你只需要将少部分的 Python 代码换成 C++就能完成。

TensorFlow 比 BNNS 或 Metal 有更多的特色。

使用 TensorFlow，你可以在模拟器上测试（Metal 通常需要在设备上运行）。

iOS 上 TensorFlow 的不足：

目前还没有 GPU 支持。TensorFlow 使用 Accelerate 框架来利用 CPU 矢量指令，速度不如在 Metal 上快。

TensorFlow 的 API 是 C++，所以你需要在面向对象的 C++中编写代码，你不能直接在 Swift 的编码。

C++的 API 比 Python 的 API 更受限制。这意味着你不能在设备上训练模型，因为目前不支持反向传播过程中所需的自动梯度计算。目前这并不太重要，因为在移动设备硬件上训练模型并不是主要需求。

TensorFlow 静态库约占 40 MB 空间。你可以通过减少支持的操作数来减小占用的空间，但这会带来很大的麻烦。这还不包括你的模型的大小。

个人认为，我现在不提倡在 iOS 上使用 TensorFlow。目前优点还不足以战胜缺点，但作为一款年轻的产品，谁知道它未来会怎样呢？

注意：如果你决定在你的 iOS 应用程序中使用 TensorFlow，你应该意识到人们很容易从你的应用程序包中复制计算图的.pb 文件，这很不安全。但这个问题并不是 TensorFlow 中所特

有的，由于冻结图文件（frozen graph file）包含了你的模型参数和计算图定义，因此你可以由此进行逆向工程。如果你的模型相当具有竞争力，你可能需要找出某种方式避免它被剽窃。

## 使用 Metal，在 GPU 上训练模型

在 iOS 应用程序中使用 TensorFlow 的一个缺点是不能使用 GPU。对于特定的项目，可能模型和数据都较小，TensorFlow 可能满足我们的需求。然而，对于更大的模型，尤其是深度学习，你可能最好使用 GPU。在 iOS 上，就意味着使用 Metal。

你仍然可以在你的 Mac 上利用 TensorFlow 训练模型，对于大模型你可以使用 GPU 甚至在云端训练，但在 iOS 上运行的推理代码使用了 Metal，而不是 TensorFlow 库。

训练好之后，我们需要导出我们学习到的参数  $W$  和  $b$ ，将其转换成 Metal 能够读取的某种格式。幸运的是，我们可以将它们保存为二进制格式的浮点数列表。

再花时间写一个 Python 脚本 `export_weights.py`；它与 `test.py` 文件很相似，用于加载计算图的定义和检查点（checkpoint）。使用如下代码：

```
W.eval().tofile("W.bin")
b.eval().tofile("b.bin")
```

`W.eval()` 用于计算当前的  $W$  值，并以 NumPy 数组返回（这与 `sess.run(W)` 所做的事情一样）。然后，我们使用 `tofile()` 将 Numpy 数组保存为二进制文件，这就是我们需要做的事情。

注：对于我们简单的分类器， $W$  是一个  $20 \times 1$  的矩阵，它仅包括了 20 个浮点数。对于更加复杂的模型，你模型的参数可能会是一个 4D 的张量。这种情况下，你将不得不对其中的一些维度做变换，因为 TensorFlow 中的数据存储形式与 Metal 的数据存储形式不一样。这可以通过简单的 `tf.transpose()` 命令来实现，但是这对于我们的分类器而言是不需要的。

让我们看看我们 logistic 回归的 Metal 版本。你可以在源代码的 VoiceMetal 文件夹中找到这个 Xcode 项目，它是用 Swift 语言写的。



logistic 回归使用了如下公式计算：

$$y\_pred = \text{sigmoid}((W * x) + b)$$

这个公式与神经网络中的全连接层中所用的公式一样，所以要用 Metal 实现我们的分类器，我们仅仅需要使用 MPSCNNFullyConnected 层。

```
let w_url = Bundle.main.url(forResource: "w", withExtension: "bin")
let b_url = Bundle.main.url(forResource: "b", withExtension: "bin")
let w_data = try! Data(contentsOf: w_url!)
let b_data = try! Data(contentsOf: b_url!)
```

首先我们将 W.bin 和 b.bin 加载到 Data 对象中；

然后我们创建全连接层：

```
let sigmoid = MPSCNNNeuronSigmoid(device: device)
let layerDesc = MPSCNNConvolutionDescriptor(
    kernelWidth: 1, kernelHeight: 1,
    inputFeatureChannels: 20, outputFeatureChannels: 1,
    neuronFilter: sigmoid)
W_data.withUnsafeBytes { W in
    b_data.withUnsafeBytes { b in
        layer = MPSCNNFullyConnected(device: device,
            convolutionDescriptor: layerDesc,
            kernelWeights: W, biasTerms: b, flags: .none)
    }
}
```

由于输入是 20 个数字，所以我决定使用一个全连接层来处理有 20 个通道的维度为 1 x 1 的「图片」。预测结果 y\_pred 是一个数值，所以全连接层仅使用一个输出通道。

输入和输出数据被储存在（与它们维度相同的）MPSImage 对象中。



```

let inputImgDesc = MPSImageDescriptor(channelFormat: .float16,
                                       width: 1, height: 1, featureChannels: 20)
let outputImgDesc = MPSImageDescriptor(channelFormat: .float16,
                                       width: 1, height: 1, featureChannels: 1)

inputImage = MPSImage(device: device, imageDescriptor: inputImgDesc)
outputImage = MPSImage(device: device, imageDescriptor: outputImgDesc)

```

与应用程序的 TensorFlow 版本一样：对于每个样本，预测方法以 20 个浮点数输入。这是完整的方法定义：

```

func predict(example: [Float]) {
    convert(example: example, to: inputImage)

    let commandBuffer = commandQueue.makeCommandBuffer()
    layer.encode(commandBuffer: commandBuffer, sourceImage: inputImage,
                destinationImage: outputImage)
    commandBuffer.commit()
    commandBuffer.waitUntilCompleted()

    let y_pred = outputImage.toFloatArray()
    print("Probability spoken by a male: \(y_pred[0])%")

    if y_pred[0] > 0.5 {
        print("Prediction: male")
    } else {
        print("Prediction: female")
    }
}

```

这是运行会话的 Metal 版本，convert(example:to:) 和 toFloatArray() 方法是加载数据和输出 MPSImage 对象的帮助器。F

这就是 Metal 应用程序！您需要在设备上运行此应用程序，因为模拟器上不支持。运行时，它应该输出以下内容：

Probability spoken by a male: 0.970215%

Prediction: male

Probability spoken by a male: 0.00568771%

Prediction: female

注：这些概率与使用 TensorFlow 预测的概率不完全相同，因为 Metal 使用的是 16 位浮点数，但是最终结果很接近

[下载天天快报，我来说两句 >](#)

## 精彩推荐

如何用TensorFlow快速搭建神经网络？来看看DeepMind新开源工具Sonnet！

热点 4天前 唯物



正在加载更多...



微信扫一扫  
下载天天快报客户端