以下资料来自英文网站

<https://tensorflow.google.cn/tutorials/keras/>

是机器翻译的结果，请对照英文资料，学习并运行所有代码，修改文字使之通顺流畅。

（使用Word的‘审阅’修订功能进行修改，保留所有修改痕迹）

学习和使用机器学习

这个笔记本系列的灵感来自[*Deep Learning with Python*](https://books.google.com/books?id=Yo3CAQAACAAJ)一书 。这些教程使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)TensorFlow的高级Python API来构建和训练深度学习模型。要了解有关在TensorFlow中使用Keras的更多信息，请参阅[TensorFlow Keras指南](https://tensorflow.google.cn/guide/keras)。

发布者注：*使用Python*深度学习使用Python语言和强大的Keras库引入*深度学习*领域。本书由Keras创作者和Google AI研究员FrançoisChollet撰写，通过直观的解释和实际例子构建对TensorFlow的认知。

要了解机器学习的基本原理和概念，请考虑参加机器学习速成 [课程](https://developers.google.cn/machine-learning/crash-course/)。其他TensorFlow和机器学习资源将在[后续步骤](https://tensorflow.google.cn/tutorials/next_steps)中列出。

1. [基本分类](https://tensorflow.google.cn/tutorials/keras/basic_classification)
2. [文字分类](https://tensorflow.google.cn/tutorials/keras/basic_text_classification)
3. [回归](https://tensorflow.google.cn/tutorials/keras/basic_regression)
4. [过度拟合和欠拟合](https://tensorflow.google.cn/tutorials/keras/overfit_and_underfit)
5. [保存和恢复模型](https://tensorflow.google.cn/tutorials/keras/save_and_restore_models)

# 训练您的第一个神经网络：基本分类

本指南训练神经网络模型，对运动鞋和衬衫等服装图像进行分类。如果您不了解所有细节，也没关系。这是一个完整的对TensorFlow程序的概述，这篇文章详细解释了技术的细节。

本指南使用[tf.keras](https://tensorflow.google.cn/guide/keras)（一个高级API），用于在TensorFlow中构建和训练模型。

# TensorFlow and tf.keras  
import tensorflow as tf  
from tensorflow import keras  
  
# Helper libraries  
import numpy as np  
import matplotlib.pyplot as plt  
  
print(tf.\_\_version\_\_)

1.11.0

## 导入Fashion MNIST数据集

本指南使用[Fashion MNIST](https://github.com/zalandoresearch/fashion-mnist)数据集，其中包含10个类别中的70,000个灰度图像。图像包括多件低分辨率（28 x 28像素）的服装，如下所示：



时尚MNIST旨在作为经典[MNIST](http://yann.lecun.com/exdb/mnist/)数据集的直接替代品，常常被看作计算机视觉机器学习计划的“Hello，World”。MNIST数据集包含手写数字（0,1,2等）的图像，格式与我们在此处使用的服装图片相同。

本指南使用Fashion MNIST进行多方面测试，因为它比普通的MNIST更具挑战性。两个数据集用于验证算法是否按预期工作，都它们相对较小，是测试和调试代码的良好起点。

我们将使用60,000个图像来训练神经网络，并用10,000个图像进行测试，以评估网络学习对图像进行分类的准确程度。您可以直接从TensorFlow访问Fashion MNIST，只需导入并加载数据即可：

fashion\_mnist = keras.datasets.fashion\_mnist  
  
(train\_images, train\_labels), (test\_images, test\_labels) = fashion\_mnist.load\_data()

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz下载数据

32768/29515 [=================================] - 0s 0us / step

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz下载数据

26427392/26421880 [==============================] - 1s 0us / step

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz下载数据

8192/5148 [============================================== =] - 0s 0us / step

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz下载数据

4423680/4422102 [==============================] - 0s 0us / step

加载数据集将返回四个NumPy数组：

* 在train\_images和train\_labels阵列的训练集模型使用学习-The数据。
* 针对测试集，数组test\_images和test\_labels数组测试模型。

图像是28x28 NumPy数组，像素值介于0到255之间。标签是一个整数数组，范围从0到9。这些对应于图像所代表的服装类别：

|  |  |
| --- | --- |
| 标签 | 类 |
| 0 | T恤/上衣 |
| 1 | 裤子 |
| 2 | 套衫 |
| 3 | 连衣裙 |
| 4 | 外套 |
| 5 | 凉鞋 |
| 6 | 衬衫 |
| 7 | 运动鞋 |
| 8 | 包包 |
| 9 | 踝靴 |

每个图像都映射到一个标签。由于类名不包含在数据集中，因此将它们存储在此处以便在绘制图像时使用：

class\_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

## 探索数据

让我们在训练模型之前探索数据集的格式。以下显示训练集中有60,000个图像，每个图像表示为28 x 28像素：

train\_images.shape

（60000,28,28）

同样，训练集中有60,000个标签：

len(train\_labels)

60000

每个标签都是0到9之间的整数：

train\_labels

数组（[9,0,0，...，3,0,5]，dtype = uint8）

测试集中有10,000个图像。同样，每个图像表示为28 x 28像素：

test\_images.shape

（10000,28,28）

测试集包含10,000个图像标签：

len(test\_labels)

10000

## 预处理数据

在训练网络之前，必须对数据进行预处理。如果您检查训练集中的第一个图像，您将看到像素值落在0到255的范围内：

plt.figure()  
plt.imshow(train\_images[0])  
plt.colorbar()  
plt.grid(False)

在把训练集应用到神经网络模型之前，我们将这些值缩放到0到1的范围。为此，需要把图像组件的数据类型从整数转换为float，并除以255。这是预处理图像的函数：

以同样的方式对训练集和测试集进行预处理非常重要：

train\_images = train\_images / 255.0  
  
test\_images = test\_images / 255.0

显示训练集中的前25个图像，并在每个图像下方显示类名，这是为了验证数据格式是否正确，我们是否已准备好构建和训练神经网络。

plt.figure(figsize=(10,10))  
for i in range(25):  
    plt.subplot(5,5,i+1)  
    plt.xticks([])  
    plt.yticks([])  
    plt.grid(False)  
    plt.imshow(train\_images[i], cmap=plt.cm.binary)  
    plt.xlabel(class\_names[train\_labels[i]])

## 搭建模型

构建神经网络需要构建模型的层，然后编译模型。

### 构建图层

神经网络的基本构建块是层。图层从提供给它们的数据中提取特征，并且希望这些图层的特征能够足够代表这些数据。

大多数深度学习包括将简单层链接在一起。大多数图层，比如[tf.keras.layers.Dense](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Dense)有在训练神经网络过程中学习到的特征。

model = keras.Sequential([  
    keras.layers.Flatten(input\_shape=(28, 28)),  
    keras.layers.Dense(128, activation=tf.nn.relu),  
    keras.layers.Dense(10, activation=tf.nn.softmax)  
])

该网络中的第一层[tf.keras.layers.Flatten](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Flatten)将图像的格式从2d阵列（28乘28像素）转换为28 \* 28 = 784像素的1d阵列。可以把这一层内容看成是将图像中层层叠叠的行展开，把它们放在一个水平高度上。该层没有要学习的参数，它只是修改数据存储结构。

在像素被展平之后，网络由[tf.keras.layers.Dense](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Dense)两层序列组成。这些是密​​集连接或完全连接的神经层。第一Dense层有128个节点（或神经元）。第二（和最后）层是10节点softmax层——返回长度为10，数值加和为1的可能性取值。每个节点都包含了一个上述可能性数值数组，十个数值分别代表这个节点属于十个类的不同可能性。

### 编译模型

在模型准备好进行训练之前，它需要更多设置。这些是在模型的编译步骤中添加的：

* 损失函数 - 这可以衡量模型在训练过程中的准确程度。我们希望最小化此函数数值，以便在正确的方向上“引导”模型。
* 优化器 - 这将根据训练数据及其损失函数优化模型的方式。
* 性能指标 - 用于监控培训和测试步骤。以下示例使用精度作为性能指标，即被正确分类的最小的图像碎片。

model.compile(optimizer=tf.train.AdamOptimizer(),   
              loss='sparse\_categorical\_crossentropy',  
              metrics=['accuracy'])

## 训练模型

训练神经网络模型需要以下步骤：

1. 将训练数据提供给模型 - 在此示例中为train\_images和train\_labels数组。
2. 该模型学会关联图像和标签。
3. 我们要求模型对测试集进行预测 - 在这个例子中，使用test\_images作为测试集。我们验证预测是否与test\_labels数组中的标签匹配。

要开始训练，需要调用model.fit方法 – 这个模型被训练来正确拟合测试集数据：

model.fit(train\_images, train\_labels, epochs=5)

大纪元1/5

60000/60000 [==============================] - 5s 90us /步 - 损失：0.5001 - acc：0.8253

大纪元2/5

60000/60000 [==============================] - 5s 88us /步 - 损失：0.3761 - acc：0.8644

大纪元3/5

60000/60000 [==============================] - 5s 89us /步 - 损失：0.3384 - acc：0.8759

大纪元4/5

60000/60000 [==============================] - 5s 89us /步 - 损失：0.3124 - acc：0.8857

大纪元5/5

60000/60000 [==============================] - 5s 86us /步 - 损失：0.2946 - acc：0.8922

损失函数大小和性能指标也被显示了出来。该模型在训练数据上达到约0.88（或88％）的准确度。

## 评估准确性

接下来，比较模型在测试数据集上的执行情况：

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)  
  
print('Test accuracy:', test\_acc)

10000/10000 [==============================] - 1s 53us /步

测试精度：0.8777

事实证明，测试数据集的准确性略低于训练数据集的准确性。训练精度和测试精度之间的差距有可能是过度拟合导致的。过度拟合是指机器学习模型在新数据上的表现比在训练数据上表现更差。

## 做出预测

通过训练模型，我们可以使用它来预测某些图像。

predictions = model.predict(test\_images)

这里，模型已经预测了测试集中每个图像的标签。我们来看看第一个预测：

predictions[0]

阵列（[4.0652740e-06,6.9819279e-08,2.5388722e-06,1.3390627e-07，

1.1847248e-07,2.9022932e-02,2.0918555e-06,6.4492501e-02，

9.1468155e-06,9.0646631e-01]，dtype = float32）

预测结果是10个数字的数组，这些数据描述了模型对图像属于10种不同服装中某一种的可能性判断的可信度。我们可以看到具有最高的置信度值的分类：

np.argmax(predictions[0])

9

因此，该模型最偏向于认定这个图像是踝靴，或class\_names[9]。我们可以检查测试标签，看看这是否正确：

test\_labels[0]

9

我们可以用图表来查看在十个类别里的情况：

def plot\_image(i, predictions\_array, true\_label, img):  
  predictions\_array, true\_label, img = predictions\_array[i], true\_label[i], img[i]  
  plt.grid(False)  
  plt.xticks([])  
  plt.yticks([])  
    
  plt.imshow(img, cmap=plt.cm.binary)  
  
  predicted\_label = np.argmax(predictions\_array)  
  if predicted\_label == true\_label:  
    color = 'blue'  
  else:  
    color = 'red'  
    
  plt.xlabel("{} {:2.0f}% ({})".format(class\_names[predicted\_label],  
                                100\*np.max(predictions\_array),  
                                class\_names[true\_label]),  
                                color=color)  
  
def plot\_value\_array(i, predictions\_array, true\_label):  
  predictions\_array, true\_label = predictions\_array[i], true\_label[i]  
  plt.grid(False)  
  plt.xticks([])  
  plt.yticks([])  
  thisplot = plt.bar(range(10), predictions\_array, color="#777777")  
  plt.ylim([0, 1])   
  predicted\_label = np.argmax(predictions\_array)  
   
  thisplot[predicted\_label].set\_color('red')  
  thisplot[true\_label].set\_color('blue')

让我们看看第0个图像、预测结果和预测数组。

i = 0  
plt.figure(figsize=(6,3))  
plt.subplot(1,2,1)  
plot\_image(i, predictions, test\_labels, test\_images)  
plt.subplot(1,2,2)  
plot\_value\_array(i, predictions,  test\_labels)

i = 12  
plt.figure(figsize=(6,3))  
plt.subplot(1,2,1)  
plot\_image(i, predictions, test\_labels, test\_images)  
plt.subplot(1,2,2)  
plot\_value\_array(i, predictions,  test\_labels)

我们使用这些预测值绘制几个图像。正确的预测标签是蓝色的，不正确的预测标签是红色的。该数字给出了预测标签的百分比占比（满分100）。请注意，即使置信度非常高，模型预测结果也可能出错。

# Plot the first X test images, their predicted label, and the true label  
# Color correct predictions in blue, incorrect predictions in red  
num\_rows = 5  
num\_cols = 3  
num\_images = num\_rows\*num\_cols  
plt.figure(figsize=(2\*2\*num\_cols, 2\*num\_rows))  
for i in range(num\_images):  
  plt.subplot(num\_rows, 2\*num\_cols, 2\*i+1)  
  plot\_image(i, predictions, test\_labels, test\_images)  
  plt.subplot(num\_rows, 2\*num\_cols, 2\*i+2)  
  plot\_value\_array(i, predictions, test\_labels)

最后，使用训练的模型对单个图像进行预测。

# Grab an image from the test dataset  
img = test\_images[0]  
  
print(img.shape)

（28,28）

[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)模型优化是为了同时预测一批的数据。因此，即使我们使用单个图像进行预测的时候，我们也需要将其放在列表中：

# Add the image to a batch where it's the only member.  
img = (np.expand\_dims(img,0))  
  
print(img.shape)

（1,28,28）

现在预测图像：

predictions\_single = model.predict(img)  
  
print(predictions\_single)

[[4.0652740e-06 6.9819272e-08 2.5388672e-06 1.3390652e-07 1.1847247e-07

2.9022938e-02 2.0918596e-06 6.4492591e-02 9.1468228e-06 9.0646625e-01]]

plot\_value\_array(0, predictions\_single, test\_labels)  
\_ = plt.xticks(range(10), class\_names, rotation=45)

model.predict返回一个列表的列表，其中每个列表对应一批数据中的每个图像。可以看到我们的图像的预测：

np.argmax(predictions\_single[0])

9

结果和以前一样，模型预测标签为9。

# 电影评论的文字分类

这篇文章使用评论文本将电影评论分类为正面或负面。这是二元或两类分类的一个例子，这是一种重要且广泛适用的机器学习问题。

我们将使用包含来自[Internet电影数据库](https://www.imdb.com/)的50,000条电影评论文本的[IMDB数据集](https://tensorflow.google.cn/api_docs/python/tf/keras/datasets/imdb)。这些分为25,000条培训评论和25,000条评审评论。培训和测试集是平衡的，这意味着它们包含相同数量的正面和负面评论。

本文使用[tf.keras](https://tensorflow.google.cn/guide/keras)（一个高级API），用于在TensorFlow中构建和训练模型。有关使用更高级的文本分类教程，请参阅[MLCC文本分类指南](https://developers.google.cn/machine-learning/guides/text-classification/)。

import tensorflow as tf  
from tensorflow import keras  
  
import numpy as np  
  
print(tf.\_\_version\_\_)

1.11.0

## 下载IMDB数据集

IMDB数据集在TensorFlow库里有。它已经通过预处理将评论（一系列的词语）转换为整数序列，其中每个整数表示字典中的特定单词。

以下代码将IMDB数据集下载到您的计算机（如果您已经下载了它，则使用缓存副本）：

imdb = keras.datasets.imdb  
  
(train\_data, train\_labels), (test\_data, test\_labels) = imdb.load\_data(num\_words=10000)

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz下载数据

17465344/17464789 [==============================] - 0s 0us / step

该参数num\_words=10000保留了训练数据中最常出现的10,000个单词，舍弃罕见的单词可以使得数据的大小在合适范围内。

## 探索数据

我们花一点时间来了解数据的格式。数据集经过预处理之后，每条评论都是一个整数数组，代表电影评论的单词。每个标签都是0或1的整数值，其中0表示负面评论，1表示正面评论。

print("Training entries: {}, labels: {}".format(len(train\_data), len(train\_labels)))

训练集数目：25000，标签数目：25000

评论文本已转换为整数，其中每个整数表示字典中的特定单词。以下是第一条评论的内容：

print(train\_data[0])

[1,14,22,16,43,530,973,1622,1385,65,458,4468,66,3941,4,173,36,256,5,25,100,43,838,112,50 ，670,2,9,35,480,284,5,150,4,172,112,167,2,336,385,39,4,172,4536,1111,17,546,38,13,447 ，4,192,50,16,6,147,2025,19,14,22,4,11920,4613,469,4,22,71,87,12,16,43,530,38,76,15 ，13,1247,4,22,17,515,17,12,16,626,18,2,5,62,386,12,8,316,8,106,5,4,2223,5244,16 ，480,66,3785,33,4,130,12,16,38,619,5,25,124,51,36,135,48,25,1415,33,6,22,12,215,28 ，77,52,5,14,407,16,82,2,8,10,10,117,5952,15,256,4,2,7,3766,5,723,36,71,43,530 ，476,26,400,317,46,7,4,2,1029,13,104,88,4,381,15,297,98,32,2071,56,26,141,6,194,7486 ，18,4,226,22,21,134,476,26,480,5,144,30,5535,18,51,36,28,224,92,25,104,4,226,65,16 ，38,1334,88,12,16,283,5,16,4472,113,103,32,15,16,5345,19,178,32]

电影评论的长度可能不同。以下代码显示了第一条和第二条评论中的字数。但是神经网络的输入内容必须是相同的长度，我们稍后需要解决此问题。

len(train\_data[0]), len(train\_data[1])

（218,189）

### 将整数转换回单词

了解如何将整数转换回文本可能很有用。在这里，我们将创建一个辅助函数来查询整数到字符串的映射：

# A dictionary mapping words to an integer index  
word\_index = imdb.get\_word\_index()  
  
# The first indices are reserved  
word\_index = {k:(v+3) for k,v in word\_index.items()}   
word\_index["<PAD>"] = 0  
word\_index["<START>"] = 1  
word\_index["<UNK>"] = 2  # unknown  
word\_index["<UNUSED>"] = 3  
  
reverse\_word\_index = dict([(value, key) for (key, value) in word\_index.items()])  
  
def decode\_review(text):  
    return ' '.join([reverse\_word\_index.get(i, '?') for i in text])

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb\_word\_index.json下载数据

1646592/1641221 [==============================] - 0s 0us / step

现在我们可以使用该decode\_review函数显示第一次审核的文本：

decode\_review(train\_data[0])

“ 这部电影只是精彩的演员位置风景故事方向每个人都非常适合他们演奏的部分你可以想象在那里罗伯特 是一个了不起的演员，现在同样是导演 父亲和我一样来自同一个苏格兰岛，所以我喜欢与这部电影真正联系的事实，整部电影中的诙谐言论非常棒，我发布的电影一出现就很精彩。 并且会推荐它给每个人观看和飞钓真是太神奇了，最后它真的很哭，它很伤心，你知道他们说什么，如果你在电影中哭，它一定是好的，这肯定也是 两个小男孩玩的 诺曼和保罗他们只是聪明的孩子往往被排除在外 列表我觉得因为那些长大的明星都是整部电影的一个重要形象，但是这些孩子都很棒，应该因为他们所做的事而受到称赞不要你认为整个故事是如此可爱因为它是真的在与我们所有人分享之后，是某人的生命“

## 数据准备

每条评论 – 也就是一串整数数组 - 必须在输入神经网络之前转换为张量。这种转换可以通过以下两种方式完成：

* 使用独热码使数组其转换为全是0或1的向量。例如，序列[3,5]将成为10,000维向量，除了索引3和5（它们是1）之外全部为零。然后，将其作为我们网络中的第一层 - 一个可以处理浮点数据向量的全连接层。然而，这种方法十分占用内存，需要num\_words \* num\_reviews大小矩阵。
* 另外一种方法是填充数组，使它们都具有相同的长度，然后创建一个大小为max\_length \* num\_reviews的整数张量。我们可以使用能够处理这种形状的张量的嵌入层作为我们网络中的第一层。

在本教程中，我们将使用第二种方法。

由于电影评论的长度必须相同，我们将使用[pad\_sequences](https://tensorflow.google.cn/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences)函数来标准化长度：

train\_data = keras.preprocessing.sequence.pad\_sequences(train\_data,  
                                                        value=word\_index["<PAD>"],  
                                                        padding='post',  
                                                        maxlen=256)  
  
test\_data = keras.preprocessing.sequence.pad\_sequences(test\_data,  
                                                       value=word\_index["<PAD>"],  
                                                       padding='post',  
                                                       maxlen=256)

我们现在看一下示例的长度：

len(train\_data[0]), len(train\_data[1])

（256,256）

并审查第一条评论填充后的：

print(train\_data[0])

[1 14 22 16 43 530 973 1622 1385 65 458 4468 66 3941

4 173 36 256 5 25 100 43 838 112 50 670 2 9

35 480 284 5 150 4 172 112 167 2 336 385 39 4

172 4536 1111 17 546 38 13 447 4 192 50 16 6 147

2025 19 14 22 4 1920 4613 469 4 22 71 87 12 16

43 530 38 76 15 13 1247 4 22 17 515 17 12 16

626 18 2 5 62 386 12 8 316 8 106 5 4 2223

5244 16 480 66 3785 33 4 130 12 16 38 619 5 25

124 51 36 135 48 25 1415 33 6 22 12 215 28 77

52 5 14 407 16 82 2 8 4 107 117 5952 15 256

4 2 7 3766 5 723 36 71 43 530 476 26 400 317

46 7 4 2 1029 13 104 88 4 381 15 297 98 32

2071 56 26 141 6 194 7486 18 4 226 22 21 134 476

26 480 5 144 30 5535 18 51 36 28 224 92 25 104

4 226 65 16 38 1334 88 12 16 283 5 16 4472 113

103 32 15 16 5345 19 178 32 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0]

## 建立模型

神经网络是通过堆叠层来创建的 - 这需要两个主要的结构性决策：

* 模型中要使用多少层？
* 每层使用多少隐藏单位？

在此示例中，输入数据由单词向量数组组成，要预测的标签是0或1。让我们为这个问题建立一个模型：

# input shape is the vocabulary count used for the movie reviews (10,000 words)  
vocab\_size = 10000  
  
model = keras.Sequential()  
model.add(keras.layers.Embedding(vocab\_size, 16))  
model.add(keras.layers.GlobalAveragePooling1D())  
model.add(keras.layers.Dense(16, activation=tf.nn.relu))  
model.add(keras.layers.Dense(1, activation=tf.nn.sigmoid))  
  
model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

嵌入（嵌入）（无，无，16）160000

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_average\_pooling1d（G1（无，16）0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

密集（密集）（无，16）272

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1（密集）（无，1）17

================================================== ===============

总参数：160,289

可训练的参数：160,289

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

这些图层按顺序堆叠以构建分类器：

1. 第一层是Embedding一层。该层采用整数编码的词汇表，并为每个单词索引查找嵌入向量。这些向量在训练模型的过程中不断调整。向量为输出数组添加一个维度，最后得到的结果是：(batch, sequence, embedding)。
2. 接下来，GlobalAveragePooling1D通过对序列维度求平均来为每个示例返回固定长度的输出向量。这使得模型以最简单的方式处理可变长度的输入。
3. 这个固定长度的输出向量通过一个带有16个隐藏单元的完全连接层进行传输。
4. 最后一层与单个输出节点密集连接。使用sigmoid激活函数，这个浮点数介于0和1之间，表示概率或置信度。

### 隐藏单元

上述模型在输入和输出之间有两个中间或“隐藏”层。输出的数量（单位，节点或神经元）是每一层特征空间的维度。换句话说，在学习内部特征的时候神经网络具有一定程度的自由度。

如果模型具有更多隐藏单元（更高维度的特征空间）或者更多层，则神经网络可以学习更复杂的特征。但是，它使神经网络的计算成本更高，并且可能导致得到的模型并非我们所想-这个模型对测试集的拟合效果不断提高，但不会提高测试数据的性能。这称为过度拟合，我们稍后会进行探讨。

### 损失函数和优化器

模型需要一个损失函数和一个用于训练的优化器。由于这是二元分类问题和模型输出概率（具有S形激活函数的单个单元层），我们将使用binary\_crossentropy损失函数。

损失函数不止这一种，例如，您可以选择mean\_squared\_error函数。但是，通常，binary\_crossentropy处理概率更好 - 它测量概率分布之间的“距离”，或者在我们的情况下，测量实际分布和预测之间的“距离”。

之后当我们探索回归问题（比如预测房子的价格）时，我们将看到如何使用另一种称为均方误差的损失函数。

现在，配置模型来使用优化器和损失函数：

model.compile(optimizer=tf.train.AdamOptimizer(),  
              loss='binary\_crossentropy',  
              metrics=['accuracy'])

## 创建验证集

在训练时，我们想要检查模型在以前没有见过的数据上的准确性。从原始训练数据中分离10,000个示例来创建验证集。（为什么不直接使用测试集？我们的目标是训练和调整我们的模型，测试数据只用来评估我们的准确性）。

x\_val = train\_data[:10000]  
partial\_x\_train = train\_data[10000:]  
  
y\_val = train\_labels[:10000]  
partial\_y\_train = train\_labels[10000:]

## 训练模型

以512个样本的小批量训练模型40个周期。这是对张量x\_train和张量y\_train中的所有样本的40次迭代。在训练期间，监控模型在验证集中的10,000个样本的损失函数和准确性：

history = model.fit(partial\_x\_train,  
                    partial\_y\_train,  
                    epochs=40,  
                    batch\_size=512,  
                    validation\_data=(x\_val, y\_val),  
                    verbose=1)

训练15000个样本，验证10000个样本

大纪元1/40

15000/15000 [==============================] - 1s 57us /步 - 损失：0.6914 - acc：0.5662 - val\_loss ：0.6886 - val\_acc：0.6416

大纪元2/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.6841 - acc：0.7016 - val\_loss ：0.6792 - val\_acc：0.6751

大纪元3/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.6706 - acc：0.7347 - val\_loss ：0.6627 - val\_acc：0.7228

大纪元4/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.6481 - acc：0.7403 - val\_loss ：0.6376 - val\_acc：0.7774

大纪元5/40

15000/15000 [==============================] - 1s 40us /步 - 损失：0.6150 - acc：0.7941 - val\_loss ：0.6017 - val\_acc：0.7862

大纪元6/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.5719 - acc：0.8171 - val\_loss ：0.5596 - val\_acc：0.7996

大纪元7/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.5230 - acc：0.8400 - val\_loss ：0.5145 - val\_acc：0.8266

大纪元8/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.4738 - acc：0.8559 - val\_loss ：0.4717 - val\_acc：0.8407

大纪元9/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.4288 - acc：0.8671 - val\_loss ：0.4343 - val\_acc：0.8500

大纪元10/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.3889 - acc：0.8794 - val\_loss ：0.4034 - val\_acc：0.8558

大纪元11/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.3558 - acc：0.8875 - val\_loss ：0.3805 - val\_acc：0.8607

大纪元12/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.3285 - acc：0.8942 - val\_loss ：0.3585 - val\_acc：0.8675

大纪元13/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.3039 - acc：0.9001 - val\_loss ：0.3432 - val\_acc：0.8707

大纪元14/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.2836 - acc：0.9056 - val\_loss ：0.3299 - val\_acc：0.8739

大纪元15/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.2661 - acc：0.9102 - val\_loss ：0.3197 - val\_acc：0.8766

大纪元16/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.2512 - acc：0.9145 - val\_loss ：0.3114 - val\_acc：0.8780

大纪元17/40

15000/15000 [==============================] - 1s 39us /步 - 损失：0.2368 - acc：0.9196 - val\_loss ：0.3046 - val\_acc：0.8800

大纪元18/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.2244 - acc：0.9235 - val\_loss ：0.2991 - val\_acc：0.8820

大纪元19/40

15000/15000 [==============================] - 1s 44us /步 - 损失：0.2129 - acc：0.9279 - val\_loss ：0.2950 - val\_acc：0.8825

大纪元20/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.2027 - acc：0.9313 - val\_loss ：0.2912 - val\_acc：0.8826

Epoch 21/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1929 - acc：0.9357 - val\_loss ：0.2884 - val\_acc：0.8836

大纪元22/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1840 - acc：0.9394 - val\_loss ：0.2868 - val\_acc：0.8843

大纪元23/40

15000/15000 [==============================] - 1s 40us /步 - 损失：0.1758 - acc：0.9429 - val\_loss ：0.2856 - val\_acc：0.8840

Epoch 24/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1677 - acc：0.9475 - val\_loss ：0.2842 - val\_acc：0.8850

大纪元25/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1606 - acc：0.9503 - val\_loss ：0.2838 - val\_acc：0.8847

大纪元26/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.1535 - acc：0.9526 - val\_loss ：0.2839 - val\_acc：0.8853

大纪元27/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.1475 - acc：0.9547 - val\_loss ：0.2851 - val\_acc：0.8841

大纪元28/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.1414 - acc：0.9571 - val\_loss ：0.2848 - val\_acc：0.8862

大纪元29/40

15000/15000 [==============================] - 1s 39us /步 - 损失：0.1356 - acc：0.9585 - val\_loss ：0.2859 - val\_acc：0.8860

大纪元30/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1307 - acc：0.9617 - val\_loss ：0.2877 - val\_acc：0.8864

大纪元31/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1248 - acc：0.9645 - val\_loss ：0.2893 - val\_acc：0.8856

大纪元32/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1202 - acc：0.9660 - val\_loss ：0.2916 - val\_acc：0.8844

大纪元33/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1149 - acc：0.9685 - val\_loss ：0.2936 - val\_acc：0.8853

Epoch 34/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1107 - acc：0.9695 - val\_loss ：0.2971 - val\_acc：0.8845

大纪元35/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.1069 - acc：0.9707 - val\_loss ：0.2987 - val\_acc：0.8854

大纪元36/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.1021 - acc：0.9731 - val\_loss ：0.3019 - val\_acc：0.8842

大纪元37/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.0984 - acc：0.9747 - val\_loss ：0.3050 - val\_acc：0.8833

大纪元38/40

15000/15000 [==============================] - 1s 42us /步 - 损失：0.0951 - acc：0.9753 - val\_loss ：0.3089 - val\_acc：0.8826

大纪元39/40

15000/15000 [==============================] - 1s 43us /步 - 损失：0.0911 - acc：0.9773 - val\_loss ：0.3111 - val\_acc：0.8829

Epoch 40/40

15000/15000 [==============================] - 1s 41us /步 - 损失：0.0876 - acc：0.9795 - val\_loss ：0.3149 - val\_acc：0.8829

## 评估模型

让我们看看模型的表现。函数将返回两个值：损失（代表模型预测错误的数量，数值越低越好）和准确性。

results = model.evaluate(test\_data, test\_labels)  
  
print(results)

25000/25000 [==============================] - 1s 36us / step

[0.33615295355796815,0.87196]

使用这种相当朴素的方法的模型准确度可以达到约87％，使用更先进的方法的话，模型准确度应该接近95％。

## 创建一段时间内准确性和损失函数的图表

model.fit()返回一个History对象，其中包含记录了训练期间发生的所有事情的字典：

history\_dict = history.history  
history\_dict.keys()

dict\_keys（['acc'，'val\_loss'，'loss'，'val\_acc']）

有四个条目：每个条目对应一个受监控的指标。我们可以使用这些指标来绘制训练集和验证集的损失函数和准确度进行比较：

import matplotlib.pyplot as plt  
  
acc = history.history['acc']  
val\_acc = history.history['val\_acc']  
loss = history.history['loss']  
val\_loss = history.history['val\_loss']  
  
epochs = range(1, len(acc) + 1)  
  
# "bo" is for "blue dot"  
plt.plot(epochs, loss, 'bo', label='Training loss')  
# b is for "solid blue line"  
plt.plot(epochs, val\_loss, 'b', label='Validation loss')  
plt.title('Training and validation loss')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
  
plt.show()

plt.clf()   # clear figure  
acc\_values = history\_dict['acc']  
val\_acc\_values = history\_dict['val\_acc']  
  
plt.plot(epochs, acc, 'bo', label='Training acc')  
plt.plot(epochs, val\_acc, 'b', label='Validation acc')  
plt.title('Training and validation accuracy')  
plt.xlabel('Epochs')  
plt.ylabel('Accuracy')  
plt.legend()  
  
plt.show()

在该图中，点表示训练集上的损失函数和准确度，实线表示验证集上的损失函数和准确度。

请注意，训练集损失函数每个周期逐渐减少，并且准确度每个周期逐渐增加。这个结果是符合使用梯度下降优化方式的预期的- 这种方式在每次迭代时都会最小化期望的数值。

但是对于验证集的损失函数和准确性来说却不是这种情况 - 它们大约迭代进行二十个周期之后达到峰值。这是过度拟合的一个例子：模型在训练数据上的表现比在以前从未见过的数据上表现得更好。此时，模型过度优化并且只会学习特定的数据特征，意味着这个模型无法推广应用到测试数据集上。

对于这种特殊情况，我们可以通过只训练二十个左右的周期来防止过度拟合。稍后，您将看到如何使用回调函数自动完成这个功能。

# 预测房价：回归

在回归问题中，我们的目标是预测连续值的输出，如价格或概率。将此与分类问题进行对比，此时我们的目标是预测离散标签（例如，包含苹果或橙子的照片）。

本文构建了一个用于预测20世纪70年代中期波士顿郊区房屋的中位数价格的模型。为此，我们将为模型提供有关郊区的一些数据点，例如犯罪率和当地财产税率。

此示例使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)API，有关详细信息，请参阅[本指南](https://tensorflow.google.cn/guide/keras)。

from \_\_future\_\_ import absolute\_import, division, print\_function  
  
import tensorflow as tf  
from tensorflow import keras  
  
import numpy as np  
  
print(tf.\_\_version\_\_)

1.11.0

## 波士顿住房价格数据集

可以在TensorFlow中直接访问此[数据集](https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html)，下载并随机使用训练集：

boston\_housing = keras.datasets.boston\_housing  
  
(train\_data, train\_labels), (test\_data, test\_labels) = boston\_housing.load\_data()  
  
# Shuffle the training set  
order = np.argsort(np.random.random(train\_labels.shape))  
train\_data = train\_data[order]  
train\_labels = train\_labels[order]

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston\_housing.npz下载数据

57344/57026 [==============================] - 0s 0us / step

### 示例和功能

这个数据集比我们迄今为止使用的其他数据集小得多：它共有506个示例，分为404个训练样例和102个测试示例：

print("Training set: {}".format(train\_data.shape))  # 404 examples, 13 features  
print("Testing set:  {}".format(test\_data.shape))   # 102 examples, 13 features

训练集：（404,13）

测试集：（102,13）

该数据集包含13个不同的功能：

1. 人均犯罪率。
2. 占地面积超过25,000平方英尺的住宅用地比例。
3. 每个城镇非零售业务的比例。
4. Charles River虚拟变量（如果管道限制河流则= 1;否则为0）。
5. 一氧化氮浓度（每千万份）。
6. 每栋住宅的平均房间数。
7. 1940年以前建造的自住单位比例。
8. 到达波士顿五个就业中心的加权距离。
9. 径向高速公路的可达性指数。
10. 每10,000美元的物业税率。
11. 城镇的学生与教师比例。
12. 1000 \*（Bk - 0.63）\*\* 2其中Bk是城镇黑人的比例。
13. 人口比例较低的概率。

这些输入数据特征中的每一个都使用不同的比例存储。某些功能由0到1之间的比例表示，其他功能的范围介于1到12之间，有些功能介于0到100之间，依此类推。这通常能够反映现实世界的情况，了解如何探索和清理此类数据是一项重要的开发技能。

**关键点：**作为建模人员和开发人员，请考虑如何使用此数据以及模型预测可能带来的潜在好处和危害。像这样的模型可能会加剧社会偏见和差异。这个模型是否与您要解决的问题相关或者是否会引起偏见？有关更多信息，请阅读[ML公平性](https://developers.google.cn/machine-learning/fairness-overview/)。

print(train\_data[0])  # Display sample features, notice the different scales

[7.8750e-02 4.5000e + 01 3.4400e + 00 0.0000e + 00 4.3700e-01 6.7820e + 00

4.1100e + 01 3.7886e + 00 5.0000e + 00 3.9800e + 02 1.5200e + 01 3.9387e + 02

6.6800e + 00]

使用[pandas](https://pandas.pydata.org/)库可以通过表格的形式显示数据集的前几行：

import pandas as pd  
  
column\_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',  
                'TAX', 'PTRATIO', 'B', 'LSTAT']  
  
df = pd.DataFrame(train\_data, columns=column\_names)  
df.head()

|  | CRIM | ZN | INDUS | CHAS | NOX | R M | 年龄 | DIS | RAD | 税 | PTRATIO | 乙 | LSTAT |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0.07875 | 45.0 | 3.44 | 0.0 | 0.437 | 6.782 | 41.1 | 3.7886 | 5 | 398.0 | 15.2 | 393.87 | 6.68 |
| 1 | 4.55587 | 0.0 | 18.10 | 0.0 | 0.718 | 3.561 | 87.9 | 1.6132 | 24.0 | 666.0 | 20.2 | 354.70 | 7.12 |
| 2 | 0.09604 | 40.0 | 6.41 | 0.0 | 0.447 | 6.854 | 42.8 | 4.2673 | 4 | 254.0 | 17.6 | 396.90 | 2.98 |
| 3 | 0.01870 | 85.0 | 4.15 | 0.0 | 0.429 | 6.516 | 27.7 | 8.5353 | 4 | 351.0 | 17.9 | 392.43 | 6.36 |
| 4 | 0.52693 | 0.0 | 6.20 | 0.0 | 0.504 | 8.725 | 83.0 | 2.8944 | 8 | 307.0 | 17.4 | 382.00 | 4.63 |

### 标签

标签是房价（单位是千美元）。（您可能会注意到20世纪70年代中期的价格。）

print(train\_labels[0:10])  # Display first 10 entries

[32。27.5 32. 23.1 50. 20.6 22.6 36.2 21.8 19.5]

## 标准化功能

建议使用不同比例和范围进行特征标准化。对于每个特征，减去其平均值并除以标准差可以得到：

# Test data is \*not\* used when calculating the mean and std  
  
mean = train\_data.mean(axis=0)  
std = train\_data.std(axis=0)  
train\_data = (train\_data - mean) / std  
test\_data = (test\_data - mean) / std  
  
print(train\_data[0])  # First training sample, normalized

[-0.39725269 1.41205707 -1.12664623 -0.25683275 -1.027385 0.72635358

-1.00016413 0.02383449 -0.51114231 -0.04753316 -1.49067405 0.41584124

-0.83648691]

如果模型可能在没有特征归一化的情况下收敛，这将使训练变得更加困难，并且所得的模型更依赖于输入的选择。

## 创建模型

让我们建立我们的模型。在这里，我们将使用Sequential模型，它具有两个密集连接的隐藏层和返回单个连续值的输出层。因为我们稍后将创建第二个模型，因此把模型构建步骤放在函数build\_model中。

def build\_model():  
  model = keras.Sequential([  
    keras.layers.Dense(64, activation=tf.nn.relu,  
                       input\_shape=(train\_data.shape[1],)),  
    keras.layers.Dense(64, activation=tf.nn.relu),  
    keras.layers.Dense(1)  
  ])  
  
  optimizer = tf.train.RMSPropOptimizer(0.001)  
  
  model.compile(loss='mse',  
                optimizer=optimizer,  
                metrics=['mae'])  
  return model  
  
model = build\_model()  
model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

密集（密集）（无，64）896

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1（密集）（无，64）4160

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2（密集）（无，1）65

================================================== ===============

总参数：5,121

可训练的参数：5,121

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 训练模型

该模型训练过500个周期，在history对象中记录模型在训练集和验证集的准确性。

# Display training progress by printing a single dot for each completed epoch  
class PrintDot(keras.callbacks.Callback):  
  def on\_epoch\_end(self, epoch, logs):  
    if epoch % 100 == 0: print('')  
    print('.', end='')  
  
EPOCHS = 500  
  
# Store training stats  
history = model.fit(train\_data, train\_labels, epochs=EPOCHS,  
                    validation\_split=0.2, verbose=0,  
                    callbacks=[PrintDot()])

.................................................. ..................................................

.................................................. ..................................................

.................................................. ..................................................

.................................................. ..................................................

.................................................. ..................................................

使用存储在history对象中的统计信息可视化模型的训练进度。我们希望使用此数据来确定在模型停止进展之前要训​​练多长时间。

import matplotlib.pyplot as plt  
  
  
def plot\_history(history):  
  plt.figure()  
  plt.xlabel('Epoch')  
  plt.ylabel('Mean Abs Error [1000$]')  
  plt.plot(history.epoch, np.array(history.history['mean\_absolute\_error']),  
           label='Train Loss')  
  plt.plot(history.epoch, np.array(history.history['val\_mean\_absolute\_error']),  
           label = 'Val loss')  
  plt.legend()  
  plt.ylim([0, 5])  
  
plot\_history(history)

该图表显示在约200个时期之后模型的改进很小。我们更新model.fit方法，以便在验证结果没有提高时自动停止训练。我们将使用一个回调来测试每个周期的训练情况。如果一定数量的周期之后没有显示出改进，就自动停止训练。

您可以[在此处](https://tensorflow.google.cn/versions/master/api_docs/python/tf/keras/callbacks/EarlyStopping)了解有关此回调的更多信息。

model = build\_model()  
  
# The patience parameter is the amount of epochs to check for improvement  
early\_stop = keras.callbacks.EarlyStopping(monitor='val\_loss', patience=20)  
  
history = model.fit(train\_data, train\_labels, epochs=EPOCHS,  
                    validation\_split=0.2, verbose=0,  
                    callbacks=[early\_stop, PrintDot()])  
  
plot\_history(history)

.................................................. ..................................................

.................................................. ...

该图显示平均误差约为2,500美元。这个好吗？好吧，当一些标签有15,000美元时，2,500美元并不是微不足道的数额。

让我们看看模型在测试集上的表现如何：

[loss, mae] = model.evaluate(test\_data, test\_labels, verbose=0)  
  
print("Testing set Mean Abs Error: ${:7.2f}".format(mae \* 1000))

测试集平均误差：2713.16美元

## 预测

最后，使用测试集中的数据预测一些房价：

test\_predictions = model.predict(test\_data).flatten()  
  
plt.scatter(test\_labels, test\_predictions)  
plt.xlabel('True Values [1000$]')  
plt.ylabel('Predictions [1000$]')  
plt.axis('equal')  
plt.xlim(plt.xlim())  
plt.ylim(plt.ylim())  
\_ = plt.plot([-100, 100], [-100, 100])

error = test\_predictions - test\_labels  
plt.hist(error, bins = 50)  
plt.xlabel("Prediction Error [1000$]")  
\_ = plt.ylabel("Count")

## 结论

本文介绍了一些处理回归问题的技巧。

* 均方误差（MSE）是常用于回归问题的损失函数（不同于分类问题）。
* 同样，用于回归的评估指标也不同于分类。常见的回归指标是平均绝对误差（MAE）。
* 当输入数据要素具有不同范围的值时，应单独缩放每个要素。
* 如果训练数据不多，则选择隐藏层较少的小型网络，以避免过度拟合。
* 早期停止是防止过度拟合的有效方法。

# 探索过度拟合和欠拟合

与往常一样，此示例中的代码将使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)API，您可以在TensorFlow [Keras指南中](https://tensorflow.google.cn/guide/keras)了解更多信息。

在之前的两个例子中 - 分类电影评论和预测住房价格 - 我们看到我们的模型对验证数据的准确性在经过多个周期的训练后会达到峰值，然后开始下降。

换句话说，我们的模型会过度拟合训练数据。学习如何处理过度拟合很重要。尽管通常可以在训练集上实现高精度，但我们真正想要的是能够很好地模拟测试数据（或之前未见过的数据）的模型。

过度拟合的反面是*欠拟合*。当测试数据仍有改进空间时，会发生欠拟合。出现这种情况的原因有很多：模型不够强大，过度正规化，或者根本没有经过足够长时间的训练。这意味着神经网络尚未学习到训练数据中的相关模式。

如果训练时间过长，模型将开始过度拟合并从训练数据中学习无法推广到测试数据模式。我们需要学会如何取得平衡。如下所述，了解如何选择训练的周期数量是一项有用的技能。

为了防止过度拟合，最好的解决方案是使用更多的训练数据。受过越多数据训练的模型自然会更容易推广。当无法再扩大数据集时，下一个最佳解决方案是使用正规化等技术。这些限制了模型可以存储的信息的数量和类型。如果一个网络只能记住少量的模式，那么优化过程将迫使它只记住最突出的模式，增大最终模型的概括性。

本文中我们将探索两种常见的正则化技术 - 权重正则化和丢弃正则化 - 并使用它们来改进我们的IMDB电影评论分类教程。

import tensorflow as tf  
from tensorflow import keras  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
print(tf.\_\_version\_\_)

1.11.0

## 下载IMDB数据集

我们不会像教程一样使用嵌入方法，而是对句子进行多热编码。该模型将很快对训练集过度拟合。它将用于证明何时发生过度拟合，以及如何解决这个问题。

对我们的列表进行多热编码意味着将它们转换为0和1的向量。具体地说，这意味着例如将序列[3, 5]转换为10,000维向量，除了索引3和5之外，它将是全零，其将是1。

NUM\_WORDS = 10000  
  
(train\_data, train\_labels), (test\_data, test\_labels) = keras.datasets.imdb.load\_data(num\_words=NUM\_WORDS)  
  
def multi\_hot\_sequences(sequences, dimension):  
    # Create an all-zero matrix of shape (len(sequences), dimension)  
    results = np.zeros((len(sequences), dimension))  
    for i, word\_indices in enumerate(sequences):  
        results[i, word\_indices] = 1.0  # set specific indices of results[i] to 1s  
    return results  
  
  
train\_data = multi\_hot\_sequences(train\_data, dimension=NUM\_WORDS)  
test\_data = multi\_hot\_sequences(test\_data, dimension=NUM\_WORDS)

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz下载数据

17465344/17464789 [==============================] - 0s 0us / step

让我们看一下生成的多热矢量。单词索引按频率排序，因此预计索引零附近有更多的1值，正如此图中看到的：

plt.plot(train\_data[0])

[]

## 过度拟合

防止过度拟合的最简单方法是减小模型的大小，即减少模型中可学习参数的数量（由层数和每层单元数决定）。在深度学习中，模型中可学习参数的数量通常被称为模型的“容量”。直观地，具有更多参数的模型将具有更多“记忆能力”，因此将能够更容易地学习到训练样本与其目标之间的字典式映射，也就是没有任何泛化能力的映射，但是在根据以往没有见过的数据做预测时这将是无用的。

始终牢记这一点：深度学习模型往往善于拟合训练数据，但真正的挑战是概括，而不是拟合。

另一方面，如果网络具有有限的记忆资源，将不会这么容易训练出映射模型。为了最大限度地减少损失，它必须学习具有更强预测能力的压缩特征。同时，如果您使模型太小，则难以适应训练数据。“太多容量”和“容量不足”之间存在平衡。

不幸的是，没有神奇的公式来确定模型的正确尺寸或架构（就层数而言，或每层的正确尺寸）。您将不得不尝试使用一系列不同的体系结构。

要找到合适的模型大小，最好从相对较少的图层和参数开始，然后开始增加图层的大小或添加新图层，直到您看到验证集损失函数规模递减为止。让我们在我们的电影评论分类的神经网络上试试这个。

我们将仅使用Dense图层创建基线模型，然后创建分别更小和更大的版本，并进行比较。

### 创建基线模型

baseline\_model = keras.Sequential([  
    # `input\_shape` is only required here so that `.summary` works.   
    keras.layers.Dense(16, activation=tf.nn.relu, input\_shape=(NUM\_WORDS,)),  
    keras.layers.Dense(16, activation=tf.nn.relu),  
    keras.layers.Dense(1, activation=tf.nn.sigmoid)  
])  
  
baseline\_model.compile(optimizer='adam',  
                       loss='binary\_crossentropy',  
                       metrics=['accuracy', 'binary\_crossentropy'])  
  
baseline\_model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

密集（密集）（无，16）160016

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1（密集）（无，16）272

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2（密集）（无，1）17

================================================== ===============

总参数：160,305

可训练的参数：160,305

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

baseline\_history = baseline\_model.fit(train\_data,  
                                      train\_labels,  
                                      epochs=20,  
                                      batch\_size=512,  
                                      validation\_data=(test\_data, test\_labels),  
                                      verbose=2)

训练25000个样本，验证25000个样本

大纪元1/20

- 5s - 损失：0.5115 - acc：0.7610 - binary\_crossentropy：0.5115 - val\_loss：0.3457 - val\_acc：0.8747 - val\_binary\_crossentropy：0.3457

大纪元2/20

- 4s - 损失：0.2539 - acc：0.9101 - binary\_crossentropy：0.2539 - val\_loss：0.2845 - val\_acc：0.8873 - val\_binary\_crossentropy：0.2845

大纪元3/20

- 4s - 损失：0.1839 - acc：0.9360 - binary\_crossentropy：0.1839 - val\_loss：0.2887 - val\_acc：0.8850 - val\_binary\_crossentropy：0.2887

大纪元4/20

- 4s - 损失：0.1474 - acc：0.9494 - binary\_crossentropy：0.1474 - val\_loss：0.3132 - val\_acc：0.8789 - val\_binary\_crossentropy：0.3132

大纪元5/20

- 4s - 损失：0.1228 - acc：0.9601 - binary\_crossentropy：0.1228 - val\_loss：0.3375 - val\_acc：0.8736 - val\_binary\_crossentropy：0.3375

大纪元6/20

- 4s - 损失：0.1021 - acc：0.9679 - binary\_crossentropy：0.1021 - val\_loss：0.3666 - val\_acc：0.8704 - val\_binary\_crossentropy：0.3666

大纪元7/20

- 4s - 损失：0.0856 - acc：0.9748 - binary\_crossentropy：0.0856 - val\_loss：0.4182 - val\_acc：0.8628 - val\_binary\_crossentropy：0.4182

大纪元8/20

- 4s - 损失：0.0723 - acc：0.9797 - binary\_crossentropy：0.0723 - val\_loss：0.4379 - val\_acc：0.8644 - val\_binary\_crossentropy：0.4379

大纪元9/20

- 4s - 损失：0.0603 - acc：0.9847 - binary\_crossentropy：0.0603 - val\_loss：0.4781 - val\_acc：0.8607 - val\_binary\_crossentropy：0.4781

大纪元10/20

- 4s - 损失：0.0501 - acc：0.9891 - binary\_crossentropy：0.0501 - val\_loss：0.5264 - val\_acc：0.8577 - val\_binary\_crossentropy：0.5264

大纪元11/20

- 4s - 损失：0.0426 - acc：0.9906 - binary\_crossentropy：0.0426 - val\_loss：0.5616 - val\_acc：0.8580 - val\_binary\_crossentropy：0.5616

大纪元12/20

- 4s - 损失：0.0336 - acc：0.9943 - binary\_crossentropy：0.0336 - val\_loss：0.6012 - val\_acc：0.8570 - val\_binary\_crossentropy：0.6012

大纪元13/20

- 4s - 损失：0.0267 - acc：0.9958 - binary\_crossentropy：0.0267 - val\_loss：0.6439 - val\_acc：0.8546 - val\_binary\_crossentropy：0.6439

大纪元14/20

- 4s - 损失：0.0208 - acc：0.9975 - binary\_crossentropy：0.0208 - val\_loss：0.6809 - val\_acc：0.8530 - val\_binary\_crossentropy：0.6809

大纪元15/20

- 4s - 损失：0.0159 - acc：0.9988 - binary\_crossentropy：0.0159 - val\_loss：0.7184 - val\_acc：0.8523 - val\_binary\_crossentropy：0.7184

大纪元16/20

- 4s - 损失：0.0125 - acc：0.9992 - binary\_crossentropy：0.0125 - val\_loss：0.7439 - val\_acc：0.8523 - val\_binary\_crossentropy：0.7439

大纪元17/20

- 4s - 损失：0.0098 - acc：0.9996 - binary\_crossentropy：0.0098 - val\_loss：0.7784 - val\_acc：0.8518 - val\_binary\_crossentropy：0.7784

大纪元18/20

- 4s - 损失：0.0077 - acc：0.9996 - binary\_crossentropy：0.0077 - val\_loss：0.8068 - val\_acc：0.8522 - val\_binary\_crossentropy：0.8068

大纪元19/20

- 4s - 损失：0.0062 - acc：0.9998 - binary\_crossentropy：0.0062 - val\_loss：0.8251 - val\_acc：0.8523 - val\_binary\_crossentropy：0.8251

大纪元20/20

- 4s - 损失：0.0051 - acc：0.9998 - binary\_crossentropy：0.0051 - val\_loss：0.8534 - val\_acc：0.8514 - val\_binary\_crossentropy：0.8534

### 创建一个较小的模型

让我们创建一个隐藏单元较少的模型，与我们刚刚创建的基线模型进行比较：

smaller\_model = keras.Sequential([  
    keras.layers.Dense(4, activation=tf.nn.relu, input\_shape=(NUM\_WORDS,)),  
    keras.layers.Dense(4, activation=tf.nn.relu),  
    keras.layers.Dense(1, activation=tf.nn.sigmoid)  
])  
  
smaller\_model.compile(optimizer='adam',  
                loss='binary\_crossentropy',  
                metrics=['accuracy', 'binary\_crossentropy'])  
  
smaller\_model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

dense\_3（密集）（无，4）40004

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4（密集）（无，4）20

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_5（密集）（无，1）5

================================================== ===============

总参数：40,029

可训练的参数：40,029

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

并使用相同的数据训练模型：

smaller\_history = smaller\_model.fit(train\_data,  
                                    train\_labels,  
                                    epochs=20,  
                                    batch\_size=512,  
                                    validation\_data=(test\_data, test\_labels),  
                                    verbose=2)

训练25000个样本，验证25000个样本

大纪元1/20

- 4s - 损失：0.5929 - acc：0.7445 - binary\_crossentropy：0.5929 - val\_loss：0.4676 - val\_acc：0.8480 - val\_binary\_crossentropy：0.4676

大纪元2/20

- 4s - 损失：0.3696 - acc：0.8842 - binary\_crossentropy：0.3696 - val\_loss：0.3464 - val\_acc：0.8764 - val\_binary\_crossentropy：0.3464

大纪元3/20

- 4s - 损失：0.2713 - acc：0.9111 - binary\_crossentropy：0.2713 - val\_loss：0.3006 - val\_acc：0.8866 - val\_binary\_crossentropy：0.3006

大纪元4/20

- 4s - 损失：0.2198 - acc：0.9268 - binary\_crossentropy：0.2198 - val\_loss：0.2858 - val\_acc：0.8869 - val\_binary\_crossentropy：0.2858

大纪元5/20

- 4s - 损失：0.1868 - acc：0.9384 - binary\_crossentropy：0.1868 - val\_loss：0.2852 - val\_acc：0.8855 - val\_binary\_crossentropy：0.2852

大纪元6/20

- 4s - 损失：0.1623 - acc：0.9464 - binary\_crossentropy：0.1623 - val\_loss：0.2872 - val\_acc：0.8853 - val\_binary\_crossentropy：0.2872

大纪元7/20

- 4s - 损失：0.1428 - acc：0.9550 - binary\_crossentropy：0.1428 - val\_loss：0.2963 - val\_acc：0.8832 - val\_binary\_crossentropy：0.2963

大纪元8/20

- 4s - 损失：0.1273 - acc：0.9598 - binary\_crossentropy：0.1273 - val\_loss：0.3092 - val\_acc：0.8799 - val\_binary\_crossentropy：0.3092

大纪元9/20

- 4s - 损失：0.1145 - acc：0.9649 - binary\_crossentropy：0.1145 - val\_loss：0.3231 - val\_acc：0.8770 - val\_binary\_crossentropy：0.3231

大纪元10/20

- 4s - 损失：0.1026 - acc：0.9698 - binary\_crossentropy：0.1026 - val\_loss：0.3381 - val\_acc：0.8748 - val\_binary\_crossentropy：0.3381

大纪元11/20

- 4s - 损失：0.0927 - acc：0.9736 - binary\_crossentropy：0.0927 - val\_loss：0.3562 - val\_acc：0.8719 - val\_binary\_crossentropy：0.3562

大纪元12/20

- 4s - 损失：0.0830 - acc：0.9774 - binary\_crossentropy：0.0830 - val\_loss：0.3736 - val\_acc：0.8699 - val\_binary\_crossentropy：0.3736

大纪元13/20

- 4s - 损失：0.0747 - acc：0.9811 - binary\_crossentropy：0.0747 - val\_loss：0.3918 - val\_acc：0.8682 - val\_binary\_crossentropy：0.3918

大纪元14/20

- 4s - 损失：0.0675 - acc：0.9843 - binary\_crossentropy：0.0675 - val\_loss：0.4135 - val\_acc：0.8657 - val\_binary\_crossentropy：0.4135

大纪元15/20

- 4s - 损失：0.0606 - acc：0.9868 - binary\_crossentropy：0.0606 - val\_loss：0.4310 - val\_acc：0.8648 - val\_binary\_crossentropy：0.4310

大纪元16/20

- 4s - 损失：0.0546 - acc：0.9888 - binary\_crossentropy：0.0546 - val\_loss：0.4520 - val\_acc：0.8636 - val\_binary\_crossentropy：0.4520

大纪元17/20

- 4s - 损失：0.0491 - acc：0.9906 - binary\_crossentropy：0.0491 - val\_loss：0.4741 - val\_acc：0.8614 - val\_binary\_crossentropy：0.4741

大纪元18/20

- 4s - 损失：0.0437 - acc：0.9922 - binary\_crossentropy：0.0437 - val\_loss：0.5019 - val\_acc：0.8597 - val\_binary\_crossentropy：0.5019

大纪元19/20

- 4s - 损失：0.0388 - acc：0.9932 - binary\_crossentropy：0.0388 - val\_loss：0.5181 - val\_acc：0.8582 - val\_binary\_crossentropy：0.5181

大纪元20/20

- 4s - 损失：0.0348 - acc：0.9943 - binary\_crossentropy：0.0348 - val\_loss：0.5407 - val\_acc：0.8574 - val\_binary\_crossentropy：0.5407

### 创建一个更大的模型

作为练习，您可以创建一个更大的模型，并查看它开始过度拟合的速度。接下来，让我们在这个基准测试中添加一个能够解决远远超出问题的范围的更大的神经网络：

bigger\_model = keras.models.Sequential([  
    keras.layers.Dense(512, activation=tf.nn.relu, input\_shape=(NUM\_WORDS,)),  
    keras.layers.Dense(512, activation=tf.nn.relu),  
    keras.layers.Dense(1, activation=tf.nn.sigmoid)  
])  
  
bigger\_model.compile(optimizer='adam',  
                     loss='binary\_crossentropy',  
                     metrics=['accuracy','binary\_crossentropy'])  
  
bigger\_model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

dense\_6（密集）（无，512）5120512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_7（密集）（无，512）262656

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_8（密集）（无，1）513

================================================== ===============

总参数：5,383,681

可训练的参数：5,383,681

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

并且，再次使用相同的数据训练模型：

bigger\_history = bigger\_model.fit(train\_data, train\_labels,  
                                  epochs=20,  
                                  batch\_size=512,  
                                  validation\_data=(test\_data, test\_labels),  
                                  verbose=2)

训练25000个样本，验证25000个样本

大纪元1/20

- 8s - 损失：0.3404 - acc：0.8564 - binary\_crossentropy：0.3404 - val\_loss：0.2946 - val\_acc：0.8808 - val\_binary\_crossentropy：0.2946

大纪元2/20

- 8s - 损失：0.1331 - acc：0.9519 - binary\_crossentropy：0.1331 - val\_loss：0.3506 - val\_acc：0.8657 - val\_binary\_crossentropy：0.3506

大纪元3/20

- 8s - 损失：0.0401 - acc：0.9889 - binary\_crossentropy：0.0401 - val\_loss：0.4434 - val\_acc：0.8681 - val\_binary\_crossentropy：0.4434

大纪元4/20

- 8s - 损失：0.0066 - acc：0.9990 - binary\_crossentropy：0.0066 - val\_loss：0.5967 - val\_acc：0.8708 - val\_binary\_crossentropy：0.5967

大纪元5/20

- 8s - 损失：9.1714e-04 - acc：1.0000 - binary\_crossentropy：9.1714e-04 - val\_loss：0.6809 - val\_acc：0.8695 - val\_binary\_crossentropy：0.6809

大纪元6/20

- 8s - 损失：2.3596e-04 - acc：1.0000 - binary\_crossentropy：2.3596e-04 - val\_loss：0.7268 - val\_acc：0.8707 - val\_binary\_crossentropy：0.7268

大纪元7/20

- 7s - 损失：1.3292e-04 - acc：1.0000 - binary\_crossentropy：1.3292e-04 - val\_loss：0.7551 - val\_acc：0.8712 - val\_binary\_crossentropy：0.7551

大纪元8/20

- 8s - 损失：9.3685e-05 - acc：1.0000 - binary\_crossentropy：9.3685e-05 - val\_loss：0.7756 - val\_acc：0.8712 - val\_binary\_crossentropy：0.7756

大纪元9/20

- 7s - 损失：7.1403e-05 - acc：1.0000 - binary\_crossentropy：7.1403e-05 - val\_loss：0.7926 - val\_acc：0.8712 - val\_binary\_crossentropy：0.7926

大纪元10/20

- 7s - 损失：5.6723e-05 - acc：1.0000 - binary\_crossentropy：5.6723e-05 - val\_loss：0.8061 - val\_acc：0.8712 - val\_binary\_crossentropy：0.8061

大纪元11/20

- 8s - 损失：4.6369e-05 - acc：1.0000 - binary\_crossentropy：4.6369e-05 - val\_loss：0.8184 - val\_acc：0.8709 - val\_binary\_crossentropy：0.8184

大纪元12/20

- 8s - 损失：3.8608e-05 - acc：1.0000 - binary\_crossentropy：3.8608e-05 - val\_loss：0.8290 - val\_acc：0.8710 - val\_binary\_crossentropy：0.8290

大纪元13/20

- 7s - 损失：3.2698e-05 - acc：1.0000 - binary\_crossentropy：3.2698e-05 - val\_loss：0.8385 - val\_acc：0.8710 - val\_binary\_crossentropy：0.8385

大纪元14/20

- 8s - 损失：2.8087e-05 - acc：1.0000 - binary\_crossentropy：2.8087e-05 - val\_loss：0.8474 - val\_acc：0.8711 - val\_binary\_crossentropy：0.8474

大纪元15/20

- 8s - 损失：2.4346e-05 - acc：1.0000 - binary\_crossentropy：2.4346e-05 - val\_loss：0.8557 - val\_acc：0.8709 - val\_binary\_crossentropy：0.8557

大纪元16/20

- 8s - 损失：2.1311e-05 - acc：1.0000 - binary\_crossentropy：2.1311e-05 - val\_loss：0.8625 - val\_acc：0.8713 - val\_binary\_crossentropy：0.8625

大纪元17/20

- 8s - 损失：1.8774e-05 - acc：1.0000 - binary\_crossentropy：1.8774e-05 - val\_loss：0.8705 - val\_acc：0.8708 - val\_binary\_crossentropy：0.8705

大纪元18/20

- 8s - 损失：1.6692e-05 - acc：1.0000 - binary\_crossentropy：1.6692e-05 - val\_loss：0.8767 - val\_acc：0.8708 - val\_binary\_crossentropy：0.8767

大纪元19/20

- 8s - 损失：1.4877e-05 - acc：1.0000 - binary\_crossentropy：1.4877e-05 - val\_loss：0.8828 - val\_acc：0.8709 - val\_binary\_crossentropy：0.8828

大纪元20/20

- 8s - 损失：1.3357e-05 - acc：1.0000 - binary\_crossentropy：1.3357e-05 - val\_loss：0.8892 - val\_acc：0.8707 - val\_binary\_crossentropy：0.8892

### 绘制训练集和验证集损失图表

实线表示训练集损失函数，虚线表示验证集损失函数（记住：较低的损失函数表示更好的模型）。在这里，较小的网络开始拟合的时间晚于基线模型（在6个时期之后而不是4个时期），并且一旦成功拟合，其性能下降得慢得多。

def plot\_history(histories, key='binary\_crossentropy'):  
  plt.figure(figsize=(16,10))  
      
  for name, history in histories:  
    val = plt.plot(history.epoch, history.history['val\_'+key],  
                   '--', label=name.title()+' Val')  
    plt.plot(history.epoch, history.history[key], color=val[0].get\_color(),  
             label=name.title()+' Train')  
  
  plt.xlabel('Epochs')  
  plt.ylabel(key.replace('\_',' ').title())  
  plt.legend()  
  
  plt.xlim([0,max(history.epoch)])  
  
  
plot\_history([('baseline', baseline\_history),  
              ('smaller', smaller\_history),  
              ('bigger', bigger\_history)])

请注意，较大的网络在仅仅一个时期之后几乎立即开始拟合，并且后面的周期里变得更加严重。网络容量越大，就能够越快地对训练数据进行建模（导致训练损失低），但过度拟合的可能性越大（导致训练和验证损失之间的差异很大）。

## 策略

### 添加权重正规化

你可能知道奥卡姆的剃刀原则：给同一个现象两种不一样的解释，最可能是正确的解释是“最简单”的解释，即做出最少量假设的解释。这也适用于神经网络学习的模型：给定一些训练数据和神经网络架构，有多组权重值（多个模型）可以解释数据，而简单模型比复杂模型更不容易过度拟合。

在这种情况下，“简单模型”是一个参数值的分布具有较少的熵的模型（或者具有较少参数的模型，如我们在上面的部分中所见）。因此，减少过度拟合的常见方法是通过强制其权重仅采用较小的值来对网络的复杂性施加约束，这使得权重值的分布更“规则”。这被称为“权重正则化”，并且通过向神经网络的损失函数调整权重大小（损失函数越大权重越大）。这个成本可以分为两个部分：

* L1正则化：按权重系数的绝对值（即权重的所谓“L1范数”）比例添加成本。
* L2正则化：按权重系数的值的平方（即，权重的所谓“L2范数”）比例添加成本。L2正则化在神经网络领域下也被称为权重衰减。不要让不同的名字所困扰：在数学上与L2正则化也被叫做权重衰减。

在[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)，权重正则化的方法的具体实现方法：权重作为正则化参数关键字传给层。现在让我们添加L2正则化。

l2\_model = keras.models.Sequential([  
    keras.layers.Dense(16, kernel\_regularizer=keras.regularizers.l2(0.001),  
                       activation=tf.nn.relu, input\_shape=(NUM\_WORDS,)),  
    keras.layers.Dense(16, kernel\_regularizer=keras.regularizers.l2(0.001),  
                       activation=tf.nn.relu),  
    keras.layers.Dense(1, activation=tf.nn.sigmoid)  
])  
  
l2\_model.compile(optimizer='adam',  
                 loss='binary\_crossentropy',  
                 metrics=['accuracy', 'binary\_crossentropy'])  
  
l2\_model\_history = l2\_model.fit(train\_data, train\_labels,  
                                epochs=20,  
                                batch\_size=512,  
                                validation\_data=(test\_data, test\_labels),  
                                verbose=2)

训练25000个样本，验证25000个样本

大纪元1/20

- 4s - 损失：0.5327 - acc：0.7878 - binary\_crossentropy：0.4928 - val\_loss：0.3877 - val\_acc：0.8784 - val\_binary\_crossentropy：0.3476

大纪元2/20

- 4s - 损失：0.3085 - acc：0.9105 - binary\_crossentropy：0.2646 - val\_loss：0.3338 - val\_acc：0.8880 - val\_binary\_crossentropy：0.2868

大纪元3/20

- 4s - 损失：0.2535 - acc：0.9298 - binary\_crossentropy：0.2041 - val\_loss：0.3345 - val\_acc：0.8864 - val\_binary\_crossentropy：0.2833

大纪元4/20

- 4s - 损失：0.2265 - acc：0.9409 - binary\_crossentropy：0.1738 - val\_loss：0.3456 - val\_acc：0.8828 - val\_binary\_crossentropy：0.2918

大纪元5/20

- 4s - 损失：0.2120 - acc：0.9482 - binary\_crossentropy：0.1567 - val\_loss：0.3596 - val\_acc：0.8799 - val\_binary\_crossentropy：0.3034

大纪元6/20

- 4s - 损失：0.1983 - acc：0.9532 - binary\_crossentropy：0.1414 - val\_loss：0.3755 - val\_acc：0.8755 - val\_binary\_crossentropy：0.3181

大纪元7/20

- 4s - 损失：0.1907 - acc：0.9561 - binary\_crossentropy：0.1323 - val\_loss：0.3921 - val\_acc：0.8730 - val\_binary\_crossentropy：0.3329

大纪元8/20

- 4s - 损失：0.1815 - acc：0.9608 - binary\_crossentropy：0.1216 - val\_loss：0.4055 - val\_acc：0.8714 - val\_binary\_crossentropy：0.3451

大纪元9/20

- 4s - 损失：0.1744 - acc：0.9627 - binary\_crossentropy：0.1136 - val\_loss：0.4249 - val\_acc：0.8697 - val\_binary\_crossentropy：0.3635

大纪元10/20

- 4s - 损失：0.1713 - acc：0.9638 - binary\_crossentropy：0.1094 - val\_loss：0.4434 - val\_acc：0.8650 - val\_binary\_crossentropy：0.3810

大纪元11/20

- 4s - 损失：0.1663 - acc：0.9656 - binary\_crossentropy：0.1034 - val\_loss：0.4568 - val\_acc：0.8638 - val\_binary\_crossentropy：0.3931

大纪元12/20

- 4s - 损失：0.1636 - acc：0.9668 - binary\_crossentropy：0.0998 - val\_loss：0.4830 - val\_acc：0.8590 - val\_binary\_crossentropy：0.4187

大纪元13/20

- 4s - 损失：0.1617 - acc：0.9674 - binary\_crossentropy：0.0965 - val\_loss：0.4843 - val\_acc：0.8602 - val\_binary\_crossentropy：0.4187

大纪元14/20

- 4s - 损失：0.1524 - acc：0.9721 - binary\_crossentropy：0.0869 - val\_loss：0.4962 - val\_acc：0.8586 - val\_binary\_crossentropy：0.4310

大纪元15/20

- 4s - 损失：0.1466 - acc：0.9752 - binary\_crossentropy：0.0815 - val\_loss：0.5105 - val\_acc：0.8592 - val\_binary\_crossentropy：0.4452

大纪元16/20

- 4s - 损失：0.1447 - acc：0.9759 - binary\_crossentropy：0.0791 - val\_loss：0.5289 - val\_acc：0.8554 - val\_binary\_crossentropy：0.4631

大纪元17/20

- 4s - 损失：0.1425 - acc：0.9762 - binary\_crossentropy：0.0767 - val\_loss：0.5411 - val\_acc：0.8554 - val\_binary\_crossentropy：0.4748

大纪元18/20

- 4s - 损失：0.1411 - acc：0.9755 - binary\_crossentropy：0.0746 - val\_loss：0.5701 - val\_acc：0.8528 - val\_binary\_crossentropy：0.5033

大纪元19/20

- 4s - 损失：0.1385 - acc：0.9772 - binary\_crossentropy：0.0715 - val\_loss：0.5639 - val\_acc：0.8555 - val\_binary\_crossentropy：0.4968

大纪元20/20

- 4s - 损失：0.1380 - acc：0.9778 - binary\_crossentropy：0.0705 - val\_loss：0.5900 - val\_acc：0.8522 - val\_binary\_crossentropy：0.5220

l2(0.001)意味着图层权重矩阵中的每个系数都会使得神经网络总损失函数增加0.001 \* weight\_coefficient\_value\*\*2。请注意，由于这种惩罚仅在训练时发生，因此在训练时这个神经网络的损失将远高于在测试时间。

这是我们的L2正规化惩罚的影响：

plot\_history([('baseline', baseline\_history),  
              ('l2', l2\_model\_history)])

正如您所看到的，即使L2正则化模型和基线模型具有相同数量的参数，但L2正则化更能减少过度拟合的可能。

### 添加丢弃层

丢弃是由Hinton和他在多伦多大学的学生开发的最有效和最常用的神经网络正则化技术之一。丢弃（应用于层）主要是指在训练期间随机丢弃多个输出层特征。假设某个给定的层，这个层在输入给定样本之后经常返回一个向量[0.2,0.5,1.3,0.8,1.1]; 如果应用被丢弃，这个向量将有几个随机分布的零条目，例如[0,0.5,1.3,0,1.1]。“丢弃率”是被设置为0的特征的比例; 它通常设置在0.2和0.5之间。在测试时，网络不会丢弃任何单元，而是将层的输出值按等同于丢弃率的比例进行缩减，以便平衡以下事实：测试时的活跃单元数大于训练时的活跃单元数。

在tf.keras中，您可以通过丢弃层将丢弃引入网络中，以便事先将其应用于层的输出。

让我们在IMDB网络中添加两个Dropout图层，看看它们在减少过度拟合方面做得如何：

dpt\_model = keras.models.Sequential([  
    keras.layers.Dense(16, activation=tf.nn.relu, input\_shape=(NUM\_WORDS,)),  
    keras.layers.Dropout(0.5),  
    keras.layers.Dense(16, activation=tf.nn.relu),  
    keras.layers.Dropout(0.5),  
    keras.layers.Dense(1, activation=tf.nn.sigmoid)  
])  
  
dpt\_model.compile(optimizer='adam',  
                  loss='binary\_crossentropy',  
                  metrics=['accuracy','binary\_crossentropy'])  
  
dpt\_model\_history = dpt\_model.fit(train\_data, train\_labels,  
                                  epochs=20,  
                                  batch\_size=512,  
                                  validation\_data=(test\_data, test\_labels),  
                                  verbose=2)

训练25000个样本，验证25000个样本

大纪元1/20

- 4s - 损失：0.6372 - acc：0.6249 - binary\_crossentropy：0.6372 - val\_loss：0.5321 - val\_acc：0.8474 - val\_binary\_crossentropy：0.5321

大纪元2/20

- 4s - 损失：0.5043 - acc：0.7721 - binary\_crossentropy：0.5043 - val\_loss：0.3879 - val\_acc：0.8752 - val\_binary\_crossentropy：0.3879

大纪元3/20

- 4s - 损失：0.4021 - acc：0.8411 - binary\_crossentropy：0.4021 - val\_loss：0.3089 - val\_acc：0.8856 - val\_binary\_crossentropy：0.3089

大纪元4/20

- 4s - 损失：0.3390 - acc：0.8752 - binary\_crossentropy：0.3390 - val\_loss：0.2821 - val\_acc：0.8890 - val\_binary\_crossentropy：0.2821

大纪元5/20

- 4s - 损失：0.2861 - acc：0.8983 - binary\_crossentropy：0.2861 - val\_loss：0.2750 - val\_acc：0.8877 - val\_binary\_crossentropy：0.2750

大纪元6/20

- 4s - 损失：0.2447 - acc：0.9154 - binary\_crossentropy：0.2447 - val\_loss：0.2794 - val\_acc：0.8876 - val\_binary\_crossentropy：0.2794

大纪元7/20

- 4s - 损失：0.2174 - acc：0.9267 - binary\_crossentropy：0.2174 - val\_loss：0.2970 - val\_acc：0.8860 - val\_binary\_crossentropy：0.2970

大纪元8/20

- 4s - 损失：0.1862 - acc：0.9366 - binary\_crossentropy：0.1862 - val\_loss：0.3034 - val\_acc：0.8841 - val\_binary\_crossentropy：0.3034

大纪元9/20

- 4s - 损失：0.1679 - acc：0.9427 - binary\_crossentropy：0.1679 - val\_loss：0.3258 - val\_acc：0.8825 - val\_binary\_crossentropy：0.3258

大纪元10/20

- 4s - 损失：0.1499 - acc：0.9491 - binary\_crossentropy：0.1499 - val\_loss：0.3374 - val\_acc：0.8813 - val\_binary\_crossentropy：0.3374

大纪元11/20

- 4s - 损失：0.1373 - acc：0.9532 - binary\_crossentropy：0.1373 - val\_loss：0.3600 - val\_acc：0.8812 - val\_binary\_crossentropy：0.3600

大纪元12/20

- 4s - 损失：0.1267 - acc：0.9561 - binary\_crossentropy：0.1267 - val\_loss：0.3735 - val\_acc：0.8792 - val\_binary\_crossentropy：0.3735

大纪元13/20

- 4s - 损失：0.1183 - acc：0.9607 - binary\_crossentropy：0.1183 - val\_loss：0.4033 - val\_acc：0.8795 - val\_binary\_crossentropy：0.4033

大纪元14/20

- 4s - 损失：0.1119 - acc：0.9617 - binary\_crossentropy：0.1119 - val\_loss：0.4225 - val\_acc：0.8792 - val\_binary\_crossentropy：0.4225

大纪元15/20

- 4s - 损失：0.1038 - acc：0.9639 - binary\_crossentropy：0.1038 - val\_loss：0.4444 - val\_acc：0.8776 - val\_binary\_crossentropy：0.4444

大纪元16/20

- 4s - 损失：0.1022 - acc：0.9640 - binary\_crossentropy：0.1022 - val\_loss：0.4432 - val\_acc：0.8784 - val\_binary\_crossentropy：0.4432

大纪元17/20

- 4s - 损失：0.0933 - acc：0.9664 - binary\_crossentropy：0.0933 - val\_loss：0.4917 - val\_acc：0.8774 - val\_binary\_crossentropy：0.4917

大纪元18/20

- 4s - 损失：0.0890 - acc：0.9690 - binary\_crossentropy：0.0890 - val\_loss：0.4953 - val\_acc：0.8766 - val\_binary\_crossentropy：0.4953

大纪元19/20

- 4s - 损失：0.0875 - acc：0.9677 - binary\_crossentropy：0.0875 - val\_loss：0.5295 - val\_acc：0.8754 - val\_binary\_crossentropy：0.5295

大纪元20/20

- 4s - 损失：0.0834 - acc：0.9700 - binary\_crossentropy：0.0834 - val\_loss：0.5309 - val\_acc：0.8749 - val\_binary\_crossentropy：0.5309

plot\_history([('baseline', baseline\_history),  
              ('dropout', dpt\_model\_history)])

添加丢弃层可明显改善基准模型。

回顾防止神经网络中过度拟合的最常见方法：

* 获取更多培训数据。
* 减少网络容量。
* 添加权重正规化。
* 添加丢弃层。

本指南还有两个重要方法没有涉及：数据增强和批量标准化。

# 保存和恢复模型

模型进度可以在培训期间和培训后保存。这意味着模型可以从上次中断的地方恢复，避免长时间的训练。保存也意味着您可以共享您的模型，而其他人可以对您的工作成功进行在创作。在发布研究模型和技术时，大多数机器学习从业者分享：

* 用于创建模型的代码
* 模型的训练权重或参数

共享此数据有助于其他人了解模型的工作原理，并使用新数据测试模型。

**注意：**小心不受信任的代码 - TensorFlow模型是代码。有关详细信息，请参阅[安全使用TensorFlow](https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md)。

### 选项

保存TensorFlow模型有多种方法，具体取决于您使用的API。本指南使用[tf.keras](https://tensorflow.google.cn/guide/keras)（一个高级API），用于在TensorFlow中构建和训练模型。有关其他方法，请参阅TensorFlow [保存和还原](https://tensorflow.google.cn/guide/saved_model)指南或[Eager中](https://tensorflow.google.cn/guide/eager" \l "object_based_saving)保存。

## 设置

### 安装和导入

安装并导入TensorFlow和依赖项：

!pip install -q h5py pyyaml

您使用的是pip版本18.0，但版本18.1可用。

您应该考虑通过'pip install --upgrade pip'命令进行升级。

### 获取示例数据集

我们将使用[MNIST数据集](http://yann.lecun.com/exdb/mnist/)来训练模型并演示保存权重。要加速这些演示运行，请仅使用前1000个示例：

from \_\_future\_\_ import absolute\_import, division, print\_function  
  
import os  
  
import tensorflow as tf  
from tensorflow import keras  
  
tf.\_\_version\_\_

'1.11.0'

(train\_images, train\_labels), (test\_images, test\_labels) = tf.keras.datasets.mnist.load\_data()  
  
train\_labels = train\_labels[:1000]  
test\_labels = test\_labels[:1000]  
  
train\_images = train\_images[:1000].reshape(-1, 28 \* 28) / 255.0  
test\_images = test\_images[:1000].reshape(-1, 28 \* 28) / 255.0

从https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz下载数据

11493376/11490434 [==============================] - 0s 0us / step

### 定义模型

让我们构建一个简单的模型，我们将用它来演示保存和加载权重。

# Returns a short sequential model  
def create\_model():  
  model = tf.keras.models.Sequential([  
    keras.layers.Dense(512, activation=tf.nn.relu, input\_shape=(784,)),  
    keras.layers.Dropout(0.2),  
    keras.layers.Dense(10, activation=tf.nn.softmax)  
  ])  
    
  model.compile(optimizer=tf.keras.optimizers.Adam(),   
                loss=tf.keras.losses.sparse\_categorical\_crossentropy,  
                metrics=['accuracy'])  
    
  return model  
  
  
# Create a basic model instance  
model = create\_model()  
model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

密集（密集）（无，512）401920

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout（Dropout）（None，512）0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1（密集）（无，10）5130

================================================== ===============

总参数：407,050

可训练的参数：407,050

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 在训练期间保存检查点

主要用例是，在培训期间和培训结束时自动保存检查点。通过这种方式，您可以使用经过训练的模型，而无需重新训练，或者在暂停地方接受训练 - 以防止训练过程中断。

[tf.keras.callbacks.ModelCheckpoint](https://tensorflow.google.cn/api_docs/python/tf/keras/callbacks/ModelCheckpoint)是执行此任务的回调。回调需要几个参数来配置检查点。

### 检查点回调使用情况

训练模型并使用ModelCheckpoint回调传递给该模型：

checkpoint\_path = "training\_1/cp.ckpt"  
checkpoint\_dir = os.path.dirname(checkpoint\_path)  
  
# Create checkpoint callback  
cp\_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint\_path,   
                                                 save\_weights\_only=True,  
                                                 verbose=1)  
  
model = create\_model()  
  
model.fit(train\_images, train\_labels,  epochs = 10,   
          validation\_data = (test\_images,test\_labels),  
          callbacks = [cp\_callback])  # pass callback to training

训练1000个样本，验证1000个样本

大纪元1/10

736/1000 [=====================> ........] - ETA：0s - 损失：1.2987 - acc：0.6345

Epoch 00001：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 487us /步 - 损失：1.1384 - acc：0.6740 - val\_loss ：0.6788 - val\_acc：0.8030

大纪元2/10

992/1000 [============================>。] - ETA：0s - 损失：0.4124 - acc：0.8881

Epoch 00002：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 218us​​ /步 - 损失：0.4159 - acc：0.8870 - val\_loss ：0.5479 - val\_acc：0.8220

大纪元3/10

672/1000 [===================> ..........] - ETA：0s - 损失：0.2909 - acc：0.9271

Epoch 00003：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 207us /步 - 损失：0.3003 - acc：0.9200 - val\_loss ：0.4593 - val\_acc：0.8490

大纪元4/10

992/1000 [============================>。] - ETA：0s - 损失：0.2030 - acc：0.9567

Epoch 00004：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 212us /步 - 损失：0.2031 - acc：0.9560 - val\_loss ：0.4272 - val\_acc：0.8660

大纪元5/10

704/1000 [====================> .........] - ETA：0s - 损失：0.1518 - acc：0.9688

Epoch 00005：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 210us /步 - 损失：0.1566 - acc：0.9690 - val\_loss ：0.4248 - val\_acc：0.8570

大纪元6/10

672/1000 [===================> ..........] - ETA：0s - 损失：0.1301 - acc：0.9688

Epoch 00006：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 209us /步 - 损失：0.1216 - acc：0.9750 - val\_loss ：0.3931 - val\_acc：0.8740

大纪元7/10

704/1000 [====================> .........] - ETA：0s - 损失：0.0821 - acc：0.9915

Epoch 00007：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 205us /步 - 损失：0.0897 - acc：0.9890 - val\_loss ：0.4251 - val\_acc：0.8570

大纪元8/10

992/1000 [============================>。] - ETA：0s - 损失：0.0684 - acc：0.9899

Epoch 00008：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 215us /步 - 损失：0.0681 - acc：0.9900 - val\_loss ：0.3910 - val\_acc：0.8590

大纪元9/10

992/1000 [============================>。] - ETA：0s - 损失：0.0485 - acc：0.9960

Epoch 00009：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 214us /步 - 损失：0.0482 - acc：0.9960 - val\_loss ：0.3973 - val\_acc：0.8720

Epoch 10/10

960/1000 [===========================> ..] - ETA：0s - 损失：0.0350 - acc：0.9990

Epoch 00010：将模型保存到training\_1 / cp.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f40a6e1f438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 211us /步 - 损失：0.0346 - acc：0.9990 - val\_loss ：0.3914 - val\_acc：0.8760

上述代码创建一个TensorFlow检查点文件集合，这些文件在每个时期结束时更新：

!ls {checkpoint\_dir}

checkpoint cp.ckpt.data-00000-of-00001 cp.ckpt.index

创建一个新的未经训练的模型。仅从权重还原模型时，必须具有与原始模型具有相同体系结构的模型。由于模型架构相同，我们可以共享权重（尽管它是模型的不同实例）。

现在重建一个新的未经训练的模型，并在测试集上进行评估。未训练模型的表现有很大的偶然性（准确率约为 10%）：

model = create\_model()  
  
loss, acc = model.evaluate(test\_images, test\_labels)  
print("Untrained model, accuracy: {:5.2f}%".format(100\*acc))

1000/1000 [==============================] - 0s 120us /步

未经训练的模型，准确度：14.20％

然后从检查点加载权重，并重新评估：

model.load\_weights(checkpoint\_path)  
loss,acc = model.evaluate(test\_images, test\_labels)  
print("Restored model, accuracy: {:5.2f}%".format(100\*acc))

1000/1000 [==============================] - 0s 40us /步

恢复模型，准确度：87.60％

### 检查点回调选项

回调提供了几个选项，可以为生成的检查点提供唯一的名称，并调整检查点创建频率。

训练一个新模型，每5个周期保存一次检查点并设置唯一名称：

# include the epoch in the file name. (uses `str.format`)  
checkpoint\_path = "training\_2/cp-{epoch:04d}.ckpt"  
checkpoint\_dir = os.path.dirname(checkpoint\_path)  
  
cp\_callback = tf.keras.callbacks.ModelCheckpoint(  
    checkpoint\_path, verbose=1, save\_weights\_only=True,  
    # Save weights, every 5-epochs.  
    period=5)  
  
model = create\_model()  
model.fit(train\_images, train\_labels,  
          epochs = 50, callbacks = [cp\_callback],  
          validation\_data = (test\_images,test\_labels),  
          verbose=0)

Epoch 00005：将模型保存到training\_2 / cp-0005.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00010：将模型保存到training\_2 / cp-0010.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00015：将模型保存到training\_2 / cp-0015.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00020：将模型保存到training\_2 / cp-0020.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00025：将模型保存到training\_2 / cp-0025.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00030：将模型保存到training\_2 / cp-0030.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00035：将模型保存到training\_2 / cp-0035.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00040：将模型保存到training\_2 / cp-0040.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00045：将模型保存到training\_2 / cp-0045.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

Epoch 00050：将模型保存到training\_2 / cp-0050.ckpt

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410cb84438>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

现在，查看生成的检查点并选择最新的检查点：

! ls {checkpoint\_dir}

checkpoint cp-0030.ckpt.data-00000-of-00001

cp-0005.ckpt.data-00000-of-00001 cp-0030.ckpt.index

cp-0005.ckpt.index cp-0035.ckpt.data-00000-of-00001

cp-0010.ckpt.data-00000-of-00001 cp-0035.ckpt.index

cp-0010.ckpt.index cp-0040.ckpt.data-00000-of-00001

cp-0015.ckpt.data-00000-of-00001 cp-0040.ckpt.index

cp-0015.ckpt.index cp-0045.ckpt.data-00000-of-00001

cp-0020.ckpt.data-00000-of-00001 cp-0045.ckpt.index

cp-0020.ckpt.index cp-0050.ckpt.data-00000-of-00001

cp-0025.ckpt.data-00000-of-00001 cp-0050.ckpt.index

CP-0025.ckpt.index

latest = tf.train.latest\_checkpoint(checkpoint\_dir)  
latest

'training\_2 / CP-0050.ckpt'

**注意：**默认的tensorflow格式仅保存最近的5个检查点。

要测试，请重置模型并加载最新的检查点：

model = create\_model()  
model.load\_weights(latest)  
loss, acc = model.evaluate(test\_images, test\_labels)  
print("Restored model, accuracy: {:5.2f}%".format(100\*acc))

1000/1000 [==============================] - 0s 96us /步

恢复模型，准确度：86.80％

## 这些文件是什么？

上述代码将权重存储到[检查点](https://tensorflow.google.cn/guide/saved_model#save_and_restore_variables)格式文件的集合中，这些文件仅包含经过训练的二进制格式的权重。检查点包含：\*包含模型权重的一个或多个分片。\*索引文件，指示哪些权重存储在哪个分片中。

如果您只在一台机器上训练模型，那么您将有一个带有后缀的分片： .data-00000-of-00001

## 手动保存权重

上文中，你看到了如何将权重加载到模型中。

手动保存权重的方法也很简单，只需要使用Model.save\_weights方法即可。

# Save the weights  
model.save\_weights('./checkpoints/my\_checkpoint')  
  
# Restore the weights  
model = create\_model()  
model.load\_weights('./checkpoints/my\_checkpoint')  
  
loss,acc = model.evaluate(test\_images, test\_labels)  
print("Restored model, accuracy: {:5.2f}%".format(100\*acc))

警告：tensorflow：此模型是使用Keras优化器（<tensorflow.python.keras.optimizers.Adam对象，位于0x7f410c725dd8>）编译的，但是使用`save\_weights`以TensorFlow格式保存。将保存模型的权重，但与TensorFlow格式的TensorFlow优化器不同，优化程序的状态将不会保存。

考虑使用TensorFlow优化器[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)。

1000/1000 [==============================] - 0s 99us /步

恢复模型，准确度：86.80％

## 保存整个模型

整个模型可以保存到包含权重值，模型配置甚至优化器配置到文件中。这使得您可以为模型设置检查点，并稍后从完全相同的状态恢复训练 – 无需访问原始代码。

在Keras中保存功能齐全的模型非常有用 - 您可以在[TensorFlow.js中](https://js.tensorflow.org/tutorials/import-keras.html)加载它们，然后在Web浏览器中训练和运行它们。

Keras使用[HDF5](https://en.wikipedia.org/wiki/Hierarchical_Data_Format)标准提供基本保存格式。对于我们来说，可以将保存的模型视为一个二进制blob。

model = create\_model()  
  
model.fit(train\_images, train\_labels, epochs=5)  
  
# Save entire model to a HDF5 file  
model.save('my\_model.h5')

大纪元1/5

1000/1000 [==============================] - 0s 395us /步 - 损失：1.1260 - acc：0.6870

大纪元2/5

1000/1000 [==============================] - 0s 135us /步 - 损失：0.4136 - acc：0.8760

大纪元3/5

1000/1000 [==============================] - 0s 138us /步 - 损失：0.2811 - acc：0.9280

大纪元4/5

1000/1000 [==============================] - 0s 153us /步 - 损失：0.2078 - acc：0.9480

大纪元5/5

1000/1000 [==============================] - 0s 154us /步 - 损失：0.1452 - acc：0.9750

现在从该文件重新创建模型：

# Recreate the exact same model, including weights and optimizer.  
new\_model = keras.models.load\_model('my\_model.h5')  
new\_model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

dense\_12（密集）（无，512）401920

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_6（Dropout）（None，512）0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_13（密集）（无，10）5130

================================================== ===============

总参数：407,050

可训练的参数：407,050

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

检查其准确性：

loss, acc = new\_model.evaluate(test\_images, test\_labels)  
print("Restored model, accuracy: {:5.2f}%".format(100\*acc))

1000/1000 [==============================] - 0s 112us / step

恢复模型，准确度：85.20％

这项技术可以如下内容：

* 权重值
* 模型配置（架构）
* 优化器配置

Keras通过检查架构来保存模型。目前，它无法保存TensorFlow优化器（来自[tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)）。使用这些优化器时，您需要在加载后重新编译模型，并且您将失去原本的优化器状态。

## 后续学习计划

这是使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)保存和加载模型的快速指南。

* 该[tf.keras指南](https://tensorflow.google.cn/guide/keras)显示如何使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)保存和加载模型。
* 请参阅在在Eager中[保存](https://tensorflow.google.cn/guide/eager" \l "object_based_saving)，了解如何在 Eager Execution 期间保存模型。
* “ [保存和还原”](https://tensorflow.google.cn/guide/saved_model)指南包含有关TensorFlow保存的低阶详细信息。

# 研究和实验

Eager Execution提供了由运行定义的命令式高级操作接口。使用自动区分编写自定义图层，前进传播和训练循环。从这些教程开始，然后阅读 [Eager Execution指南](https://tensorflow.google.cn/guide/eager)。

1. Eager Execution
2. [自动微分和](https://tensorflow.google.cn/tutorials/eager/automatic_differentiation)梯度记录
3. [自定义训练：基础知识](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/notebooks/custom_training.ipynb)
4. [自定义图层](https://tensorflow.google.cn/tutorials/eager/custom_layers)
5. [自定义训练：演示](https://tensorflow.google.cn/tutorials/eager/custom_training_walkthrough)

# Eager execution基础

这是使用TensorFlow的入门教程。它将涵盖：

* 所需导入的包
* 创建和使用张量
* 使用GPU加速
* 数据集

## 导入TensorFlow

首先，导入tensorflow模块并启用eager execution。Eager execution为TensorFlow提供了一个更加具有互动性的前端，我们将在稍后讨论其中的详细说明。

import tensorflow as tf  
  
tf.enable\_eager\_execution()

## 张量

张量是一个多维数组。与NumPy ndarray对象类似，Tensor对象具有自己的数据类型和形状。此外，向量可以保存在加速器（如GPU）内存中。TensorFlow提供了丰富的操作库（[tf.add](https://tensorflow.google.cn/api_docs/python/tf/add)，[tf.matmul](https://tensorflow.google.cn/api_docs/python/tf/matmul)，[tf.linalg.inv](https://tensorflow.google.cn/api_docs/python/tf/linalg/inv)等），它们可以用于使用和生成向量。这些操作可以帮助我们把向量自动转换传统Python类。例如：

print(tf.add(1, 2))  
print(tf.add([1, 2], [3, 4]))  
print(tf.square(5))  
print(tf.reduce\_sum([1, 2, 3]))  
print(tf.encode\_base64("hello world"))  
  
# Operator overloading is also supported  
print(tf.square(2) + tf.square(3))

tf.Tensor（3，shape =（），dtype = int32）

tf.Tensor（[4 6]，shape =（2，），dtype = int32）

tf.Tensor（25，shape =（），dtype = int32）

tf.Tensor（6，shape =（），dtype = int32）

tf.Tensor（b'aGVsbG8gd29ybGQ'，shape =（），dtype = string）

tf.Tensor（13，shape =（），dtype = int32）

每个向量都有一个形状和数据类型

x = tf.matmul([[1]], [[2, 3]])  
print(x.shape)  
print(x.dtype)

（1,2）

<dtype：'int32'>

NumPy阵列和TensorFlow张量之间最明显的区别是：

1. 张量可以由加速器内存（如GPU，TPU）加速。
2. 张量是不可改变的。

### NumPy兼容性

TensorFlow张量和NumPy 队列之间的转换非常简单，如：

* TensorFlow操作自动将NumPy 队列转换为张量。
* NumPy操作自动将张量转换为NumPy 队列。

通过调用.numpy()方法，可以将张量显式转换为NumPy 队列。这种转换通常很简单，因为这个操作将尽可能的使得队列和张量共享底层内存。但是，共享底层内存并不总是可行的，因为张量可能在GPU内存中，而NumPy阵列只能由主机内存支持，因此这种情况下的转换将涉及从GPU到主机内存的复制过程。

import numpy as np  
  
ndarray = np.ones([3, 3])  
  
print("TensorFlow operations convert numpy arrays to Tensors automatically")  
tensor = tf.multiply(ndarray, 42)  
print(tensor)  
  
  
print("And NumPy operations convert Tensors to numpy arrays automatically")  
print(np.add(tensor, 1))  
  
print("The .numpy() method explicitly converts a Tensor to a numpy array")  
print(tensor.numpy())

TensorFlow操作自动将numpy数组转换为Tensors

tf.Tensor（

[42。42. 42.]

[42。42. 42.]

[42。42. 42.]]，shape =（3,3），dtype = float64）

NumPy操作会自动将Tensors转换为numpy数组

[43。43.。]

[43。43.。]

[43。43. 43.]

.nu​​mpy（）方法显式地将Tensor转换为numpy数组

[42。42. 42.]

[42。42. 42.]

[42。42. 42.]]

## GPU加速

使用GPU进行计算可以加速许多TensorFlow操作。在没有任何注释的情况下，TensorFlow会自动决定是使用GPU还是CPU进行操作（如有必要，还可以复制CPU和GPU内存之间的张量）。由操作产生的张量由执行这个操作的内存来支持。例如：

x = tf.random\_uniform([3, 3])  
  
print("Is there a GPU available: "),  
print(tf.test.is\_gpu\_available())  
  
print("Is the Tensor on GPU #0:  "),  
print(x.device.endswith('GPU:0'))

有可用的GPU：

假

是GPU＃0上的Tensor：

假

### 设备名称

该Tensor.device属性提供管理张量内容的设备名称列表。这个名称列表包含很多信息，例如正在执行此程序的主机的网络ip地址以及该主机中的设备等。这是执行分布式TensorFlow程序所必需的。如果张量在第N主机上的GPU上执行，则字符串以GPU:<N>结束。

### 显式设备放置

TensorFlow中的术语“placement”指的是如何为各个执行操作分配合适的装置。如上所述，当没有提供明确的指示时，TensorFlow会自动决定执行操作的设备，并在需要时将Tensors复制到该设备。但是，可以使用[tf.device](https://tensorflow.google.cn/api_docs/python/tf/device)内容管理器显式要求TensorFlow操作运行在特定设备上。例如：

def time\_matmul(x):  
  %timeit tf.matmul(x, x)  
  
# Force execution on CPU  
print("On CPU:")  
with tf.device("CPU:0"):  
  x = tf.random\_uniform([1000, 1000])  
  assert x.device.endswith("CPU:0")  
  time\_matmul(x)  
  
# Force execution on GPU #0 if available  
if tf.test.is\_gpu\_available():  
  with tf.device("GPU:0"): # Or GPU:1 for the 2nd GPU, GPU:2 for the 3rd etc.  
    x = tf.random\_uniform([1000, 1000])  
    assert x.device.endswith("GPU:0")  
    time\_matmul(x)

在CPU上：

每回路7.09 ms±151μs（平均值±标准偏差，7次运行，每次100次循环）

## 数据集

本节演示了使用 [tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)API来构建给模型提供数据的通道。它涵盖两个方面：

* 创建一个Dataset。
* 用eager execution对Dataset迭代。

我们建议使用Datasets API从简单，可重复使用的部分构建高性能，复杂的输入通道，来为模型的培训或评估循环提供支持。

如果您熟悉TensorFlow图，则在Dataset启用eager execution时，构建对象的API 与不使用eager execution时保持完全相同，但迭代数据集元素的过程稍微简单一些。您可以对[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)对象使用Python迭代，而不需要显式创建[tf.data.Iterator](https://tensorflow.google.cn/api_docs/python/tf/data/Iterator)对象。因此，在启用eager execution时，[TensorFlow指南中](https://tensorflow.google.cn/guide/datasets)对迭代器的讨论[无关紧要](https://tensorflow.google.cn/guide/datasets)。

### 创建一个源 数据集

使用其中一个函数创建源数据集[Dataset.from\_tensors](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#from_tensors)，[Dataset.from\_tensor\_slices](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#from_tensor_slices)或使用从[TextLineDataset](https://tensorflow.google.cn/api_docs/python/tf/data/TextLineDataset)或等文件读取的对象[TFRecordDataset](https://tensorflow.google.cn/api_docs/python/tf/data/TFRecordDataset)。有关更多信息，请参阅[TensorFlow指南](https://tensorflow.google.cn/guide/datasets#reading_input_data)。

ds\_tensors = tf.data.Dataset.from\_tensor\_slices([1, 2, 3, 4, 5, 6])  
  
# Create a CSV file  
import tempfile  
\_, filename = tempfile.mkstemp()  
  
with open(filename, 'w') as f:  
  f.write("""Line 1  
Line 2  
Line 3  
  """)  
  
ds\_file = tf.data.TextLineDataset(filename)

### 应用转换

使用如转换函数[map](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#map)，[batch](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#batch)，[shuffle](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#shuffle)等来转换应用到数据集的记录。有关详细信息，请参阅[API文档tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)。

ds\_tensors = ds\_tensors.map(tf.square).shuffle(2).batch(2)  
  
ds\_file = ds\_file.batch(2)

### 迭代

当启用eager execution时，Dataset对象支持迭代。如果您熟悉在TensorFlow图中使用Dataset，那就不需要Dataset.make\_one\_shot\_iterator()或get\_next()。

print('Elements of ds\_tensors:')  
for x in ds\_tensors:  
  print(x)  
  
print('\nElements in ds\_file:')  
for x in ds\_file:  
  print(x)

ds\_tensors的元素：

tf.Tensor（[1 9]，shape =（2，），dtype = int32）

tf.Tensor（[4 16]，shape =（2，），dtype = int32）

tf.Tensor（[25 36]，shape =（2，），dtype = int32）

ds\_file中的元素：

tf.Tensor（[b'Line 1'b'Line 2']，shape =（2，），dtype = string）

tf.Tensor（[b'Line 3'b'']，shape =（2，），dtype = string）

# 自动求导和梯度磁带

在上一个教程中，我们介绍了Tensors和操作。在本教程中，我们将介绍[自动差异化](https://en.wikipedia.org/wiki/Automatic_differentiation)，这是优化机器学习模型的关键技术。

## 初始化

import tensorflow as tf  
tf.enable\_eager\_execution()  
  
tfe = tf.contrib.eager # Shorthand for some symbols

## 函数的导数

TensorFlow提供用于自动分类的API - 计算函数的导数。更接近模仿数学的方式是将计算封装在Python函数中，记为f，使用tfe.gradients\_function创建一个函数来计算f与其相关的参数导数。如果您熟悉使用[autograd](https://github.com/HIPS/autograd)对numpy函数进行求导的话，那么您将对这部分内容很熟悉。例如：

from math import pi  
  
def f(x):  
  return tf.square(tf.sin(x))  
  
assert f(pi/2).numpy() == 1.0  
  
  
# grad\_f will return a list of derivatives of f  
# with respect to its arguments. Since f() has a single argument,  
# grad\_f will return a list with a single element.  
grad\_f = tfe.gradients\_function(f)  
assert tf.abs(grad\_f(pi/2)[0]).numpy() < 1e-7

### 高阶导数

可以使用相同的API任意多次求导：

def f(x):  
  return tf.square(tf.sin(x))  
  
def grad(f):  
  return lambda x: tfe.gradients\_function(f)(x)[0]  
  
x = tf.lin\_space(-2\*pi, 2\*pi, 100)  # 100 points between -2π and +2π  
  
import matplotlib.pyplot as plt  
  
plt.plot(x, f(x), label="f")  
plt.plot(x, grad(f)(x), label="first derivative")  
plt.plot(x, grad(grad(f))(x), label="second derivative")  
plt.plot(x, grad(grad(grad(f)))(x), label="third derivative")  
plt.legend()  
plt.show()

## 梯度磁带

每个可求导的TensorFlow操作都具有相关的梯度函数。例如，梯度函数tf.square(x)返回的将是函数2.0 \* x。要计算用户定义函数的梯度（f(x)如上例所示），TensorFlow首先“记录”应用于计算函数输出的所有操作。我们将此记录称为“磁带”。然后，它使用该磁带和与每个基元操作相关联的梯度函数，以使用[反向模式](https://en.wikipedia.org/wiki/Automatic_differentiation)求导来计算用户定义函数的梯度。

由于操作是在执行时记录的，因此Python需要控制流程（例如使用ifs和whiles）：

def f(x, y):  
  output = 1  
  # Must use range(int(y)) instead of range(y) in Python 3 when  
  # using TensorFlow 1.10 and earlier. Can use range(y) in 1.11+  
  for i in range(int(y)):  
    output = tf.multiply(output, x)  
  return output  
  
def g(x, y):  
  # Return the gradient of `f` with respect to it's first parameter  
  return tfe.gradients\_function(f)(x, y)[0]  
  
assert f(3.0, 2).numpy() == 9.0   # f(x, 2) is essentially x \* x  
assert g(3.0, 2).numpy() == 6.0   # And its gradient will be 2 \* x  
assert f(4.0, 3).numpy() == 64.0  # f(x, 3) is essentially x \* x \* x  
assert g(4.0, 3).numpy() == 48.0  # And its gradient will be 3 \* x \* x

有时并不方便将需要的计算封装到函数中。例如，如果希望输出函数计算中的中间值的梯度的时候。在这种情况下，更详细明确的[tf.GradientTape](https://tensorflow.google.cn/api_docs/python/tf/GradientTape)方法是有用的。A内的所有计算都被[tf.GradientTape](https://tensorflow.google.cn/api_docs/python/tf/GradientTape) “记录”。

例如：

x = tf.ones((2, 2))  
    
# a single t.gradient() call when the bug is resolved.  
with tf.GradientTape(persistent=True) as t:  
  t.watch(x)  
  y = tf.reduce\_sum(x)  
  z = tf.multiply(y, y)  
  
# Use the same tape to compute the derivative of z with respect to the  
# intermediate value y.  
dz\_dy = t.gradient(z, y)  
assert dz\_dy.numpy() == 8.0  
  
# Derivative of z with respect to the original input tensor x  
dz\_dx = t.gradient(z, x)  
for i in [0, 1]:  
  for j in [0, 1]:  
    assert dz\_dx[i][j].numpy() == 8.0

### 高阶梯度

GradientTape记录内容管理器内部的操作以自动求导。如果在该函数中也计算梯度，那么内容管理器也记录梯度计算。因此，完全相同的API也适用于高阶梯度。例如：

x = tf.constant(1.0)  # Convert the Python 1.0 to a Tensor object  
  
with tf.GradientTape() as t:  
  with tf.GradientTape() as t2:  
    t2.watch(x)  
    y = x \* x \* x  
  # Compute the gradient inside the 't' context manager  
  # which means the gradient computation is differentiable as well.  
  dy\_dx = t2.gradient(y, x)  
d2y\_dx2 = t.gradient(dy\_dx, x)  
  
assert dy\_dx.numpy() == 3.0  
assert d2y\_dx2.numpy() == 6.0

## 下一步

在本教程中，我们介绍了TensorFlow中的导数计算。有了这个，我们就拥有了构建和训练神经网络所需的足够基本知识。

# 自定义训练：基础知识

在上一个教程中，我们介绍了用于自动求导的TensorFlow API，这是机器学习的基本构建块。在本教程中，我们将使用先前教程中介绍的TensorFlow基本知识来进行一些简单的机器学习练习。

TensorFlow还包括一个更高级别的神经网络API（[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)），它提供了有用的抽象来减少样板。我们强烈建议使用神经网络的大家使用更高级别的API。但是，在这个简短的教程中，我们从第一原理开始介绍神经网络训练，以建立坚实的基础。

## 建立

import tensorflow as tf  
  
tfe = tf.contrib.eager  
  
tf.enable\_eager\_execution()

## 变量

TensorFlow中的张量是不可变的无状态对象。然而，机器学习模型需要具有变化阶段：随着模型训练，计算预测的相同代码应该随着时间的推移而得到不同结果（希望具有越来越低的损失！）。因为Python是一种状态编程语言，所以如果需要在计算过程中记录进行更改的状态，Python是个很好的选择：

# Using python state  
x = tf.zeros([10, 10])  
x += 2  # This is equivalent to x = x + 2, which does not mutate the original  
        # value of x  
print(x)

tf.Tensor（

[[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2。2. 2. 2. 2. 2. 2. 2. 2. 2.]]，shape =（10,10），dtype = float32）

但是，TensorFlow内置了状态操作，这些操作通常比您直接使用Python表示更令容易操作。例如，为了表示模型​​中的权重，使用TensorFlow变量通常是方便有效的。

变量是一个存储值的对象，当在TensorFlow计算中使用时，它将隐式地从该存储值中读取。有些操作（[tf.assign\_sub](https://tensorflow.google.cn/api_docs/python/tf/assign_sub)，[tf.scatter\_update](https://tensorflow.google.cn/api_docs/python/tf/scatter_update)等）操纵存储在TensorFlow变量中的值。

v = tf.Variable(1.0)  
assert v.numpy() == 1.0  
  
# Re-assign the value  
v.assign(3.0)  
assert v.numpy() == 3.0  
  
# Use `v` in a TensorFlow operation like tf.square() and reassign  
v.assign(tf.square(v))  
assert v.numpy() == 9.0

在计算梯度时，会自动跟踪使用变量的数值。对于表示嵌入TensorFlow的变量，默认会进行稀疏更新，这样可以提高计算效率和内存效率。

使用变量也是一种快速让代码的读者知道某段状态是否可变的方法。

## 示例：拟合线性模型

现在，使用几个概念--- Tensor，GradientTape，Variable---建立和训练一个简单的模型。通常涉及几个步骤：

1. 定义模型。
2. 定义损失函数。
3. 获取训练数据。
4. 运行数据训练并使用“优化器”调整变量以拟合数据。

在本教程中，我们将介绍一个简单线性模型的简单示例：f(x) = x \* W + b它有两个变量 - W和b。我们综合数据集使得最终训练模型为W = 3.0和b = 2.0。

### 定义模型

让我们定义一个简单的类来封装变量和计算。

class Model(object):  
  def \_\_init\_\_(self):  
    # Initialize variable to (5.0, 0.0)  
    # In practice, these should be initialized to random values.  
    self.W = tfe.Variable(5.0)  
    self.b = tfe.Variable(0.0)  
      
  def \_\_call\_\_(self, x):  
    return self.W \* x + self.b  
    
model = Model()  
  
assert model(3.0).numpy() == 15.0

### 定义损失函数

损失函数测量给定输入的模型输出与期望输出的匹配程度。我们使用标准L2损失来测试。

def loss(predicted\_y, desired\_y):  
  return tf.reduce\_mean(tf.square(predicted\_y - desired\_y))

### 获取训练数据

让我们用合成一些带噪音的训练数据。

TRUE\_W = 3.0  
TRUE\_b = 2.0  
NUM\_EXAMPLES = 1000  
  
inputs  = tf.random\_normal(shape=[NUM\_EXAMPLES])  
noise   = tf.random\_normal(shape=[NUM\_EXAMPLES])  
outputs = inputs \* TRUE\_W + TRUE\_b + noise

在我们训练模型之前，让我们想象一下模型现在的样子。我们将用红色表示模型的预测，用蓝色表示训练数据。

import matplotlib.pyplot as plt  
  
plt.scatter(inputs, outputs, c='b')  
plt.scatter(inputs, model(inputs), c='r')  
plt.show()  
  
print('Current loss: '),  
print(loss(model(inputs), outputs).numpy())

当前损失：

9.172191

### 定义训练循环

我们现在拥有我们的网络和训练数据。我们现在将使用训练数据来更新模型的变量（W和b）达到训练的目的，以便使用[梯度下降](https://en.wikipedia.org/wiki/Gradient_descent)来减少损失。在[tf.train.Optimizer](https://tensorflow.google.cn/api_docs/python/tf/train/Optimizer)中可以找到梯度下降方案的许多办法。我们强烈建议使用这些实现，但本着从第一原则构建的精神，在这个特定的例子中，我们将自己实现基本的数学。

def train(model, inputs, outputs, learning\_rate):  
  with tf.GradientTape() as t:  
    current\_loss = loss(model(inputs), outputs)  
  dW, db = t.gradient(current\_loss, [model.W, model.b])  
  model.W.assign\_sub(learning\_rate \* dW)  
  model.b.assign\_sub(learning\_rate \* db)

最后，让我们通过反复训练数据运行，看看W和b变化。

model = Model()  
  
# Collect the history of W-values and b-values to plot later  
Ws, bs = [], []  
epochs = range(10)  
for epoch in epochs:  
  Ws.append(model.W.numpy())  
  bs.append(model.b.numpy())  
  current\_loss = loss(model(inputs), outputs)  
  
  train(model, inputs, outputs, learning\_rate=0.1)  
  print('Epoch %2d: W=%1.2f b=%1.2f, loss=%2.5f' %  
        (epoch, Ws[-1], bs[-1], current\_loss))  
  
# Let's plot it all  
plt.plot(epochs, Ws, 'r',  
         epochs, bs, 'b')  
plt.plot([TRUE\_W] \* len(epochs), 'r--',  
         [TRUE\_b] \* len(epochs), 'b--')  
plt.legend(['W', 'b', 'true W', 'true\_b'])  
plt.show()  
 

时期0：W = 5.00 b = 0.00，损失= 9.17219

时期1：W = 4.59b = 0.40，损失= 6.23177

时期2：W = 4.26 b = 0.72，损失= 4.35631

时期3：W = 4.00 b = 0.97，损失= 3.16009

时期4：W = 3.79b = 1.17，损失= 2.39710

时期5：W = 3.62b = 1.33，损失= 1.91045

时期6：W = 3.49b = 1.46，损失= 1.60004

时期7：W = 3.38b = 1.57，损失= 1.40206

时期8：W = 3.30b = 1.65，损失= 1.27577

时期9：W = 3.23b = 1.72，损失= 1.19522

## 下一步

在本教程中，我们介绍了Variables，并使用到目前为止讨论的TensorFlow基本知识构建并训练了一个简单的线性模型。

从理论上讲，这几乎是您使用TensorFlow进行机器学习研究所需要的全部内容。在实践中，特别是对于神经网络，更高级别的API [tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)会更方便，因为它提供更高级别的构建块（称为“层”），保存和恢复状态的实用程序、一套损失函数、一套优化策略等等

# 自定义图层

我们建议使用[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)作为构建神经网络的高级API。也就是说，大多数TensorFlow API都可以在eager execution时使用。

import tensorflow as tf  
tfe = tf.contrib.eager  
  
tf.enable\_eager\_execution()

## 图层：有用的常用操作集

大多数情况下，在为机器学习模型编写代码时，您希望在比单个操作和单个变量操作更高的抽象级别上操作。

许多机器学习模型可以表达为相对简单的图层的组合和堆叠，TensorFlow提供了许多常用图层，方便您从头开始或者组合许多现有的图层创建自己的应用程序特定图层的方法。

TensorFlow包括完整的[Keras](https://keras.io/)API在tf.keras包里，建立自己的模型时Keras层是非常有用的。

# In the tf.keras.layers package, layers are objects. To construct a layer,  
# simply construct the object. Most layers take as a first argument the number  
# of output dimensions / channels.  
layer = tf.keras.layers.Dense(100)  
# The number of input dimensions is often unnecessary, as it can be inferred  
# the first time the layer is used, but it can be provided if you want to   
# specify it manually, which is useful in some complex models.  
layer = tf.keras.layers.Dense(10, input\_shape=(None, 5))

可以在[文档中](https://tensorflow.google.cn/api_docs/python/tf/keras/layers)看到预先存在的图层的完整列表。它包括Dense（完全连接层），Conv2D，LSTM，BatchNormalization，Dropout等等。

# To use a layer, simply call it.  
layer(tf.zeros([10, 5]))

# Layers have many useful methods. For example, you can inspect all variables  
# in a layer by calling layer.variables. In this case a fully-connected layer  
# will have variables for weights and biases.  
layer.variables

[，

]

# The variables are also accessible through nice accessors  
layer.kernel, layer.bias

（，

）

## 实现自定义图层

实现自定义层的最佳方法是扩展tf.keras.Layer类：您可以在\* \_\_init\_\_其中执行所有与输入无关的初始化。您可以在\* build其中了解输入张量的形状，并可以执行其余的初始化。你可以从\* call 中了解做运算的部位。

请注意，您不必额外使用build来调用创建变量，您也可以在创建模型的时候直接变量\_\_init\_\_。但是， build方法好处在于它可以根据图层将要操作的输入的形状实现后期变量创建。使用\_\_init\_\_创建变量意味着一开始就需要明确指定创建变量所需的形状。

class MyDenseLayer(tf.keras.layers.Layer):  
  def \_\_init\_\_(self, num\_outputs):  
    super(MyDenseLayer, self).\_\_init\_\_()  
    self.num\_outputs = num\_outputs  
      
  def build(self, input\_shape):  
    self.kernel = self.add\_variable("kernel",   
                                    shape=[input\_shape[-1].value,   
                                           self.num\_outputs])  
      
  def call(self, input):  
    return tf.matmul(input, self.kernel)  
    
layer = MyDenseLayer(10)  
print(layer(tf.zeros([10, 5])))  
print(layer.variables)

tf.Tensor（

[[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0 0. 0

[0。0. 0 0. 0 0. 0 0. 0. 0.]]，shape =（10,10），dtype = float32）

[<tf.Variable'my\_dense\_layer / kernel：0'shape =（5,10）dtype = float32，numpy =

数组[（0.5552308,0.15261263，-0.50605863，-0.28906852,0.19212198，

-0.03093904,0.2988543,0.22826433，-0.1261915，-0.10878032]，

[0.09125155，-0.39299783，-0.21365738,0.33756983，-0.41095817，

-0.3370653,0.5044417，-0.6214354,0.07145101，-0.5890037]，

[0.03779471,0.49143927,0.26023144，-0.21885717，-0.5952207，

0.44505948，-0.2458311，-0.21887362,0.41163248，-0.63018376]，

[0.34816158，-0.24168107，-0.483324,0.5507062,0.4923038，

-0.5348824,0.4177242,0.40392286，-0.10790956，-0.255685]，

[-0.3073315，-0.28438622，-0.21212474,0.1558457,0.17647111，

0.2545814，-0.4331643，-0.3602615，-0.3484104,0.3704632]]，

D型= FLOAT32）>]

请注意，您不必等到build调用创建变量，您也可以在其中创建变量\_\_init\_\_。

尽可能使用标准层，这样整体代码更易于阅读和维护，因为其他读者将熟悉标准层的行为。如果你想使用tf.keras.layers或tf.contrib.layers中不存在的图层，请考虑提交[github问题，](http://github.com/tensorflow/tensorflow/issues/new)或者更好的是，向我们发送请求！

## 模型：组成图层

机器学习模型中许多有趣的层是通过组合现有层来实现的。例如，resnet中的每个残余块是卷积，批量标准化和快捷方式的组合。

创建包含其他图层的图层时使用的主类是tf.keras.Model。具体实现过程是通过继承自tf.keras.Model。

class ResnetIdentityBlock(tf.keras.Model):  
  def \_\_init\_\_(self, kernel\_size, filters):  
    super(ResnetIdentityBlock, self).\_\_init\_\_(name='')  
    filters1, filters2, filters3 = filters  
  
    self.conv2a = tf.keras.layers.Conv2D(filters1, (1, 1))  
    self.bn2a = tf.keras.layers.BatchNormalization()  
  
    self.conv2b = tf.keras.layers.Conv2D(filters2, kernel\_size, padding='same')  
    self.bn2b = tf.keras.layers.BatchNormalization()  
  
    self.conv2c = tf.keras.layers.Conv2D(filters3, (1, 1))  
    self.bn2c = tf.keras.layers.BatchNormalization()  
  
  def call(self, input\_tensor, training=False):  
    x = self.conv2a(input\_tensor)  
    x = self.bn2a(x, training=training)  
    x = tf.nn.relu(x)  
  
    x = self.conv2b(x)  
    x = self.bn2b(x, training=training)  
    x = tf.nn.relu(x)  
  
    x = self.conv2c(x)  
    x = self.bn2c(x, training=training)  
  
    x += input\_tensor  
    return tf.nn.relu(x)  
  
      
block = ResnetIdentityBlock(1, [1, 2, 3])  
print(block(tf.zeros([1, 2, 3, 3])))  
print([x.name for x in block.variables])

tf.Tensor（

[[[[0。0. 0.]

[0。0. 0.]

[0。0. 0.]]

[[0。0. 0.]

[0。0. 0.]

[0。0. 0.]]]]，shape =（1,2,3,3），dtype = float32）

['resnet\_identity\_block / conv2d / kernel：0'，'resnet\_identity\_block / conv2d / bias：0'，'resnet\_identity\_block / batch\_normalization / gamma：0'，'resnet\_identity\_block / batch\_normalization / beta：0'，'resnet\_identity\_block / conv2d\_1 / kernel：0' ，'resnet\_identity\_block / conv2d\_1 / bias：0'，'resnet\_identity\_block / batch\_normalization\_1 / gamma：0'，'resnet\_identity\_block / batch\_normalization\_1 / beta：0'，'resnet\_identity\_block / conv2d\_2 / kernel：0'，'resnet\_identity\_block / conv2d\_2 / bias：0' ，'resnet\_identity\_block / batch\_normalization\_2 / gamma：0'，'resnet\_identity\_block / batch\_normalization\_2 / beta：0'，'resnet\_identity\_block / batch\_normalization / moving\_mean：0'，'resnet\_identity\_block / batch\_normalization / moving\_variance：0'，'resnet\_identity\_block / batch\_normalization\_1 / moving\_mean：0'，'resnet\_identity\_block / batch\_normalization\_1 / moving\_variance：0'，'resnet\_identity\_block / batch\_normalization\_2 / moving\_mean：0'，'resnet\_identity\_block / batch\_normalization\_2 / moving\_variance：0']

然而，在很多时候，组成许多层的模型只是将一层接一层地称为一层。这可以使用tf.keras.Sequential在非常少的代码中完成。

 my\_seq = tf.keras.Sequential([tf.keras.layers.Conv2D(1, (1, 1)),  
                               tf.keras.layers.BatchNormalization(),  
                               tf.keras.layers.Conv2D(2, 1,   
                                                      padding='same'),  
                               tf.keras.layers.BatchNormalization(),  
                               tf.keras.layers.Conv2D(3, (1, 1)),  
                               tf.keras.layers.BatchNormalization()])  
my\_seq(tf.zeros([1, 2, 3, 3]))

## 下一步

现在，您可以回到之前的教程并调整线性回归示例，使用更好的结构化图层和模型进行尝试。

# 自定义训练：演示

本指南使用机器学习按物种对鸢尾花进行分类。它使用TensorFlow的Eager Execution 来：1。构建模型，2。在示例数据上训练此模型，以及3.使用模型对未知数据进行预测。

## TensorFlow编程

本指南使用以下高级TensorFlow概念：

* 启用 [Eager Execution](https://tensorflow.google.cn/guide/eager) 开发环境，
* 使用[数据集API](https://tensorflow.google.cn/guide/datasets)导入数据，
* 使用TensorFlow的[Keras API](https://keras.io/getting-started/sequential-model-guide/)构建模型和图层。

本教程采用了许多TensorFlow程序类似的结构：

1. 导入并解析数据集。
2. 选择模型类型。
3. 训练模型。
4. 评估模型的效果。
5. 使用训练好的模型进行预测。

## 安装程序

### 配置导入和Eager Execution

导入所需的Python模块 - 包括TensorFlow – 然后针对此程序启用Eager Execution。借助Eager Execution使TensorFlow立即评估各项操作，返回具体值，而不是创建稍后执行的[计算图](https://tensorflow.google.cn/guide/graphs)。如果您习惯使用REPL或python交互式控制台，使用Eager Execution 您会用起来得心应手。[Tensorlow> = 1.8](https://tensorflow.google.cn/install/)提供了可以执行Eager Execution。

一旦启用了Eager Execution，就无法在同一程序中停用。有关详细信息，请参阅 [Eager Execution指南](https://tensorflow.google.cn/guide/eager)。

from \_\_future\_\_ import absolute\_import, division, print\_function  
  
import os  
import matplotlib.pyplot as plt  
  
import tensorflow as tf  
import tensorflow.contrib.eager as tfe  
  
tf.enable\_eager\_execution()  
  
print("TensorFlow version: {}".format(tf.VERSION))  
print("Eager execution: {}".format(tf.executing\_eagerly()))

TensorFlow版本：1.11.0

Eager Execution：是的

## 鸢尾花分类问题

想象一下，你是一名植物学家，正在寻找一种能够对所发现的每株鸢尾花进行自动归类的方法。机器学习提供了许多算法来对花进行统计分类。例如，复杂的机器学习程序可以根据照片对花进行分类。我们的要求并不高- 我们将根据[萼片](https://en.wikipedia.org/wiki/Sepal)和[花瓣](https://en.wikipedia.org/wiki/Petal)的长度和宽度测量来对鸢尾花进行分类。

鸢尾花属包含约300种，但我们的计划仅对以下三种进行分类：

* 山鸢尾

维吉尼亚鸢尾

* 变色鸢尾

|  |
| --- |
|  |
| 图 [1.Iris](https://commons.wikimedia.org/wiki/User:Dlanglois)[setosa](https://commons.wikimedia.org/w/index.php?curid=170298)（由[Radomil](https://commons.wikimedia.org/wiki/User:Radomil)，CC BY-SA 3.0），[Iris versicolor](https://commons.wikimedia.org/w/index.php?curid=248095)，（由[Dlanglois](https://commons.wikimedia.org/wiki/User:Dlanglois)，CC BY-SA 3.0）和[Iris virginica](https://www.flickr.com/photos/33397993@N05/3352169862)（由[Frank Mayfield](https://www.flickr.com/photos/33397993@N05)，CC BY-SA 2.0）。 |

幸运的是，有人已经用萼片和花瓣测量值创建了[120个鸢尾花](https://en.wikipedia.org/wiki/Iris_flower_data_set)的[数据集](https://en.wikipedia.org/wiki/Iris_flower_data_set)。这是一个在入门级机器学习分类问题中经常使用的经典数据集。

## 导入和解析训练数据集

下载数据集文件并将其转换为可供此Python程序使用的结构。

### 下载数据集

使用[tf.keras.utils.get\_file](https://tensorflow.google.cn/api_docs/python/tf/keras/utils/get_file)函数下载训练数据集文件。这将返回下载文件的文件路径。

train\_dataset\_url = "http://download.tensorflow.org/data/iris\_training.csv"  
  
train\_dataset\_fp = tf.keras.utils.get\_file(fname=os.path.basename(train\_dataset\_url),  
                                           origin=train\_dataset\_url)  
  
print("Local copy of the dataset file: {}".format(train\_dataset\_fp))

从http://download.tensorflow.org/data/iris\_training.csv下载数据

8192/2194 [============================================== ================================================== ================] - 0s 0us / step

数据集文件的本地副本：/root/.keras/datasets/iris\_training.csv

### 检查数据

此数据集iris\_training.csv是一个纯文本文件，其中存储了逗号分隔的表格数据（CSV）。使用head -n5命令在查看前 5 个条目：

!head -n5 {train\_dataset\_fp}

120,4，setosa,versicolor,virginica

6.4,2.8,5.6,2.2,2

5.0,2.3,3.3,1.0,1

4.9,2.5,4.5,1.7,2

4.9,3.1,1.5,0.1,0

从数据集的此视图中可以发现以下内容：

1. 第一行是数据集的信息的标题：
   * 总共有120个例子。每个示例都有四个功能和和一个标签名称，标签名称有三种可能。
2. 后面的行是数据记录，每个样本各占一行，其中：
   * 前四个字段是[*特征*](https://developers.google.cn/machine-learning/glossary/#feature)：即是示例的特征。这里，字段包含表示花卉测量值的浮点数。
   * 最后一列是[*标签*](https://developers.google.cn/machine-learning/glossary/#label)：即我们想要预测的值。对于此数据集，它是与花名称对应的整数值0,1或2。

我们在代码中写出来：

# column order in CSV file  
column\_names = ['sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'species']  
  
feature\_names = column\_names[:-1]  
label\_name = column\_names[-1]  
  
print("Features: {}".format(feature\_names))  
print("Label: {}".format(label\_name))

特点：['sepal\_length'，'sepal\_width'，'petal\_length'，'petal\_width']

标签：物种

每个标签都与字符串名称相关联（例如，“setosa”），但机器学习通常依赖于数值。因此将标签号映射到命名来展示结果，例如：

* 0：山鸢尾
* 1：变色鸢尾
* 2：维吉尼亚鸢尾

有关功能和标签的更多信息，请参阅[机器学习速成课程ML术语部分](https://developers.google.cn/machine-learning/crash-course/framing/ml-terminology)。

class\_names = ['Iris setosa', 'Iris versicolor', 'Iris virginica']

### 创建一个 [tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)

TensorFlow的[数据集API](https://tensorflow.google.cn/guide/datasets)可处理在向模型加载数据时遇到的许多常见情况。这是一个高级API，用于读取数据并将其转换为用于培训的格式。有关详细信息，请参阅“ [数据集快速入门”指南](https://tensorflow.google.cn/get_started/datasets_quickstart)。

由于数据集是CSV格式的文本文件，因此请使用[make\_csv\_dataset](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/make_csv_dataset)函数将数据解析为合适的格式。由于此函数为训练模型生成数据，因此默认行为是对数据（shuffle=True, shuffle\_buffer\_size=10000）进行随机处理，并无限期重复数据集（num\_epochs=None）。我们还设置了[batch\_size](https://developers.google.cn/machine-learning/glossary/#batch_size)参数。

batch\_size = 32  
  
train\_dataset = tf.contrib.data.make\_csv\_dataset(  
    train\_dataset\_fp,  
    batch\_size,   
    column\_names=column\_names,  
    label\_name=label\_name,  
    num\_epochs=1)

make\_csv\_dataset 函数返回 (features, label) 对的 tf.data.Dataset，其中 features 是一个字典：{'feature\_name': value}

启用eager Execution后，这些Dataset对象便可迭代的。我们来看看一批特征：

features, labels = next(iter(train\_dataset))  
  
features

OrderedDict（[（ 'sepal\_length'，

）

（ 'sepal\_width'，

）

（ 'petal\_length'，

）

（ 'petal\_width'，

）]）

请注意，相似的特征被组合在一起，即分为一批。每个样本行的字段都附加到相应的特征数组。更改batch\_size以设置存储在这些特征数组中的样本数。

绘制该批次中的几个特征后，就会开始看到一些集群现象：

plt.scatter(features['petal\_length'],  
            features['sepal\_length'],  
            c=labels,  
            cmap='viridis')  
  
plt.xlabel("Petal length")  
plt.ylabel("Sepal length");

要简化模型构建步骤，请创建一个函数以特征字典重新打包为具有形状的单个数组：(batch\_size, num\_features)。

此函数使用[tf.stack](https://tensorflow.google.cn/api_docs/python/tf/stack)方法，该方法从张量列表中获取值，并创建指定维度组合张量。

def pack\_features\_vector(features, labels):  
  """Pack the features into a single array."""  
  features = tf.stack(list(features.values()), axis=1)  
  return features, labels

然后使用[tf.data.Dataset.map](https://tensorflow.google.cn/api_docs/python/tf/data/dataset/map)方法将features每(features,label)打包到训练数据集中：

train\_dataset = train\_dataset.map(pack\_features\_vector)

现在的特征元素Dataset是具有(batch\_size, num\_features)形状的数组。我们来看几个例子：

features, labels = next(iter(train\_dataset))  
  
print(features[:5])

tf.Tensor（

[[6.3 3.4 5.6 2.4]

[5.3 3.7 1.5 0.2]

[7.2 3.6 6.1 2.5]

[4.5 2.3 1.3 0.3]

[5.8 2.7 4.1 1.]]，shape =（5,4），dtype = float32）

## 选择模型类型

### 为何选型？

一个[*模型*](https://developers.google.cn/machine-learning/crash-course/glossary#model)是功能和标签之间的关系。对于鸢尾花分类问题，该模型定义了萼片和花瓣测量与预测的鸢尾花种类之间的关系。一些简单的模型可以用几行代数来描述，但是复杂的机器学习模型具有很多难以概括的参数。

你能否在不使用机器学习的情况下确定四种特征与鸢尾花物种之间的关系？也就是说，您可以使用传统的编程技术（例如，很多条件语句）来创建模型吗？也许 - 如果你对数据集进行了足够长的分析，或许可以确定特定物种的花瓣和萼片测量值之间的关系。但是在更复杂的数据集上，这变得很困难 – 甚至可以说是不可能的。良好的机器学习方法可以为您确定模型。如果您将足够的代表性示例提供给正确的机器学习模型类型，程序将为您找出他们之间的关系。

### 选择模型

我们需要选择要训练的模型。模型具有许多类型，挑选合适的类型需要一定的经验支持。本教程使用神经网络来解决鸢尾花分类问题。[*神经网络*](https://developers.google.cn/machine-learning/glossary/#neural_network)可以找到特征和标签之间的复杂关系。神经网络是一个高度结构化的图形，其中包含一个或多个[*隐藏层*](https://developers.google.cn/machine-learning/glossary/#hidden_layer)。每个隐藏层由一个或多个[*神经元组成*](https://developers.google.cn/machine-learning/glossary/#neuron)。神经网络有多种类别，该程序使用的是密集型神经网络，也称为[*完全连接的神经网络*](https://developers.google.cn/machine-learning/glossary/#fully_connected_layer)：一个层中的神经元将从上一层中的每个神经元获取输入连接。例如，图2说明了一个密集的神经网络，包括一个输入层，两个隐藏层和一个输出层：

|  |
| --- |
|  |
| 图2.具有特征，隐藏层和预测的神经网络。 |

当图 2 中的模型经过训练并获得无标签样本后，它会产生 3 个预测结果：相应鸢尾花属于指定品种的可能性。这种预测称为[*推理*](https://developers.google.cn/machine-learning/crash-course/glossary#inference)。对于这个示例，输出预测的总和为1.0。在图2中，该预测分解为：山鸢尾为0.02，变色鸢尾为0.95，虹膜锦葵为0.03。这意味着模型以95％的概率预测示例花是变色额鸢尾花。

### 使用Keras创建模型

TensorFlow [tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras) API是创建模型和图层的首选方式。通过该 API，您可以轻松地构建模型并进行实验，而将所有部分连接在一起的复杂工作则由 Keras 处理。

该[tf.keras.Sequential](https://tensorflow.google.cn/api_docs/python/tf/keras/Sequential)模型是层的线性堆栈。该模型的构造函数会采用一系列层实例。在本实例中，两个[Dense](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Dense)层各有10个节点，一个输出层有3个代表我们预测标签的节点。第一层的input\_shape参数对应于数据集中的要素数，并且是必需的。

model = tf.keras.Sequential([  
  tf.keras.layers.Dense(10, activation=tf.nn.relu, input\_shape=(4,)),  # input shape required  
  tf.keras.layers.Dense(10, activation=tf.nn.relu),  
  tf.keras.layers.Dense(3)  
])

的[*激活函数*](https://developers.google.cn/machine-learning/crash-course/glossary#activation_function)确定在层中的每个节点的输出形状。这些非线性关系很重要 - 没有它们，模型将等同于单个层。有许多[可用的激活](https://tensorflow.google.cn/api_docs/python/tf/keras/activations)，但隐藏层通常使用 ReLU。

隐藏层和神经元的理想数量取决于问题和数据集。像机器学习的其他方面一样，选择神经网络的最佳形状需要知识和实验经验。根据经验，增加隐藏层和神经元的数量通常会创建一个更强大的模型，这需要更多数据才能有效地进行训练。

### 使用模型

让我们快速了解一下这个模型如何处理一批特征：

predictions = model(features)  
predictions[:5]

这里，每个样本针对每个类别返回一个对数。

要将这些对数转换为每个类的概率，需要使用[softmax](https://developers.google.cn/machine-learning/crash-course/glossary#softmax)函数：

tf.nn.softmax(predictions[:5])

对每个类别执行[tf.argmax](https://tensorflow.google.cn/api_docs/python/tf/argmax)运算可得出预测的类别索引。但是，该模型还没有被训练过，所以这些都不是很好的预测。

print("Prediction: {}".format(tf.argmax(predictions, axis=1)))  
print("    Labels: {}".format(labels))

预测：[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

标签：[2 0 2 0 1 1 1 2 1 2 0 2 1 0 1 1 0 1 2 1 1 1 0 0 1 2 0 0 2 2 0]

## 训练模型

[*训练*](https://developers.google.cn/machine-learning/crash-course/glossary#training)是机器学习的阶段，在此阶段中，模型会逐渐得到优化，也就是说，模型会了解数据集。目标是充分了解训练数据集的结构，以便对没见过的数据进行预测。如果您对训练数据集了解得太多，那么预测仅适用于它已经看到的数据，并且不会被推广。这个问题被称为[*过度拟合*](https://developers.google.cn/machine-learning/crash-course/glossary#overfitting) - 就好比将答案死记硬背下来，而不去理解问题的解决方式。

鸢尾花分类问题是[*监督机器学习的*](https://developers.google.cn/machine-learning/glossary/#supervised_machine_learning)一个例子：模型是从包含标签的示例中加以训练。在[*无监督的机器学习中*](https://developers.google.cn/machine-learning/glossary/#unsupervised_machine_learning)，示例不包含标签。相反，模型通常在特征中找到规律。

### 定义损失和梯度函数

培训和评估阶段都需要计算模型的[*损失*](https://developers.google.cn/machine-learning/crash-course/glossary#loss)。这可以衡量模型预测与所需标签的关系，换句话说，模型的执行情况有多糟糕。我们希望最小化或优化这个值。

我们的模型将使用[tf.keras.losses.categorical\_crossentropy](https://tensorflow.google.cn/api_docs/python/tf/losses/sparse_softmax_cross_entropy)函数计算其损失，该函数获取模型的类别概率预测结果和预期标签，然后返回样本中的平均损失。

def loss(model, x, y):  
  y\_ = model(x)  
  return tf.losses.sparse\_softmax\_cross\_entropy(labels=y, logits=y\_)  
  
  
l = loss(model, features, labels)  
print("Loss test: {}".format(l))

损失测试：1.800889015197754

使用[tf.GradientTape](https://tensorflow.google.cn/api_docs/python/tf/GradientTape) context计算用于优化模型的*梯度*。有关此更多示例，请参阅 [Eager Execution指南](https://tensorflow.google.cn/guide/eager)。

def grad(model, inputs, targets):  
  with tf.GradientTape() as tape:  
    loss\_value = loss(model, inputs, targets)  
  return loss\_value, tape.gradient(loss\_value, model.trainable\_variables)

### 创建优化器

[*优化*](https://developers.google.cn/machine-learning/crash-course/glossary#optimizer)器会将计算出的梯度应用于模型中的变量，以最小化loss函数。您可以将损失函数视为曲面（参见图3），我们希望通过四处走动直到找到它的最低点。梯度指向最高速上升的方向，因此我们将沿相反的方向向下移动。通过迭代方式计算每批的损失和梯度，以在训练期间调整模型。模型会逐渐找到权重和偏差的最佳组合，从而最小化损失。损失越低，模型的预测越好。

|  |
| --- |
|  |
| 图3.在3D空间中随时间可视化的优化算法。 （来源：[斯坦福大学CS231n级](http://cs231n.github.io/neural-networks-3/)，麻省理工学院执照，图片来源：[Alec Radford](https://twitter.com/alecrad)） |

TensorFlow有许多可用于训练的[优化算法](https://tensorflow.google.cn/api_guides/python/train)。该模型使用[tf.train.GradientDescentOptimizer](https://tensorflow.google.cn/api_docs/python/tf/train/GradientDescentOptimizer)，他可以实现[*随机梯度下降*](https://developers.google.cn/machine-learning/crash-course/glossary#gradient_descent)（SGD）算法。learning\_rate用于设置每次迭代（向下行走）的步长。这是一个超参数，您通常会调整这个参数以获得更好的结果。

我们设置优化器和global\_step计数器：

optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.01)  
  
global\_step = tf.train.get\_or\_create\_global\_step()

我们将使用它来计算单个优化步骤：

loss\_value, grads = grad(model, features, labels)  
  
print("Step: {}, Initial Loss: {}".format(global\_step.numpy(),  
                                          loss\_value.numpy()))  
  
optimizer.apply\_gradients(zip(grads, model.variables), global\_step)  
  
print("Step: {},         Loss: {}".format(global\_step.numpy(),  
                                          loss(model, features, labels).numpy()))

步骤：0，初始损失：1.800889015197754

步骤：1，损失：1.6618385314941406

### 训练循环

一切准备就绪后，就可以开始训练模型了！训练循环将数据集示例提供给模型，以帮助模型做出更好的预测。以下代码块可以设置这些训练步骤：

1. 迭代每个周期。通过一次数据集即为一个周期。
2. 在一个周期内，迭代训练集Dataset中的每个样本，获取它的特征（x）和标签（y）。
3. 根据样本的特征进行预测，并比较预测结果和标签。衡量预测结果的不准确性，并使用所得的值计算模型的损失和梯度。
4. 使用optimizer更新模型的变量。
5. 跟踪一些统计数据进行可视化。
6. 对每个周期重复执行以上步骤。

num\_epochs变量是遍历数据集集合的次数。与直觉相反的是，长时间训练模型并不能保证模型变得更好。num\_epochs是一个可以调整的[*超参数*](https://developers.google.cn/machine-learning/glossary/#hyperparameter)。选择正确的数字通常需要经验和实验积累。

## Note: Rerunning this cell uses the same model variables  
  
# keep results for plotting  
train\_loss\_results = []  
train\_accuracy\_results = []  
  
num\_epochs = 201  
  
for epoch in range(num\_epochs):  
  epoch\_loss\_avg = tfe.metrics.Mean()  
  epoch\_accuracy = tfe.metrics.Accuracy()  
  
  # Training loop - using batches of 32  
  for x, y in train\_dataset:  
    # Optimize the model  
    loss\_value, grads = grad(model, x, y)  
    optimizer.apply\_gradients(zip(grads, model.variables),  
                              global\_step)  
  
    # Track progress  
    epoch\_loss\_avg(loss\_value)  # add current batch loss  
    # compare predicted label to actual label  
    epoch\_accuracy(tf.argmax(model(x), axis=1, output\_type=tf.int32), y)  
  
  # end epoch  
  train\_loss\_results.append(epoch\_loss\_avg.result())  
  train\_accuracy\_results.append(epoch\_accuracy.result())  
    
  if epoch % 50 == 0:  
    print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,  
                                                                epoch\_loss\_avg.result(),  
                                                                epoch\_accuracy.result()))

Epoch 000：损失：1.603，准确度：35.000％

大纪元050：损失：0.493，准确度：79.167％

Epoch 100：损失：0.301，准确度：95.833％

时代150：损失：0.178，准确度：98.333％

Epoch 200：损失：0.135，准确度：98.333％

### 可视化损失函数随着时间的推移而变化的情况

虽然输出模型的训练过程很有帮助，但看到这一进展通常会更有帮助。[TensorBoard](https://tensorflow.google.cn/guide/summaries_and_tensorboard)与TensorFlow封装在一起是一个很好的可视化工具，我们可以使用该matplotlib模块创建基本图表。

解释这些图表需要一些经验，但我们希望看到损失下降，准确性上升。

fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))  
fig.suptitle('Training Metrics')  
  
axes[0].set\_ylabel("Loss", fontsize=14)  
axes[0].plot(train\_loss\_results)  
  
axes[1].set\_ylabel("Accuracy", fontsize=14)  
axes[1].set\_xlabel("Epoch", fontsize=14)  
axes[1].plot(train\_accuracy\_results);

## 评估模型的有效性

现在模型已经过训练，我们可以对其性能进行一些评估。

评估意味着确定模型是否能有效地进行预测。为了确定模型在鸢尾花分类方面的有效性，将一些萼片和花瓣测量值传递给模型，并要求模型预测它们代表的鸢尾花种类，然后将模型的预测与实际标签进行比较。例如，如果模型对一半输入样本的品种预测正确，则*准确率*有0.5。图4显示了一个效果更好的模型，其中5个预测里有4个判断正确，准确度为80％：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 示例功能 | | | | 标签 | 模型预测 |
| 5.9 | 3.0 | 4.3 | 1.5 | 1 | 1 |
| 6.9 | 3.1 | 5.4 | 2.1 | 2 | 2 |
| 5.1 | 3.3 | 1.7 | 0.5 | 0 | 0 |
| 6 | 3.4 | 4.5 | 1.6 | 1 | 2 |
| 5.5 | 2.5 | 4 | 1.3 | 1 | 1 |
| 图4.一个80％准确的鸢尾花分类器。 | | | | | |

### 设置测试数据集

评估模型类似于训练模型。最大的区别是样本来自单独的[*测试集*](https://developers.google.cn/machine-learning/crash-course/glossary#test_set)而不是训练集。为了公平地评估模型的有效性，用于评估模型的示例必须与用于训练模型的示例不同。

测试设置Dataset类似于培训设置Dataset。下载CSV文本文件并解析该值，然后给它进行随机化处理：

test\_url = "http://download.tensorflow.org/data/iris\_test.csv"  
  
test\_fp = tf.keras.utils.get\_file(fname=os.path.basename(test\_url),  
                                  origin=test\_url)

从http://download.tensorflow.org/data/iris\_test.csv下载数据

8192/573 [============================================== ================================================== ================================================== ================================================== ================================================== ================================================== ================================================== ================================================== ================================] - 0s 0us / step

test\_dataset = tf.contrib.data.make\_csv\_dataset(  
    test\_fp,  
    batch\_size,   
    column\_names=column\_names,  
    label\_name='species',  
    num\_epochs=1,  
    shuffle=False)  
  
test\_dataset = test\_dataset.map(pack\_features\_vector)

### 在数据集上评估测试模型

与训练阶段不同，该模型仅评估测试数据的单个周期。在下面的代码单元格中，我们遍历测试集中的每个示例，并将模型的预测与实际标签进行比较。这是为了测量模型在整个测试集中的准确性。

test\_accuracy = tfe.metrics.Accuracy()  
  
for (x, y) in test\_dataset:  
  logits = model(x)  
  prediction = tf.argmax(logits, axis=1, output\_type=tf.int32)  
  test\_accuracy(prediction, y)  
  
print("Test set accuracy: {:.3%}".format(test\_accuracy.result()))

测试设定精度：96.667％

例如，我们可以看到最后一批数据，模型预测是正确的：

tf.stack([y,prediction],axis=1)

## 使用训练的模型进行预测

我们已经训练了一个模型并且“证明”它对鸢尾花物种进行分类是有效的 - 但不是完美的。现在让我们使用训练有素的模型对[未标记的例子](https://developers.google.cn/machine-learning/glossary/#unlabeled_example)做出一些预测; 也就是说，这些例子包含特征但不包含标签。

在现实生活中，未标记的示例可能来自许多不同的来源，包括应用程序，CSV文件和数据源。目前，我们将手动提供三个未标记的示例来预测其标签。回想一下，标签编号会映射到一个指定的表示法：

* 0：山鸢尾
* 1：变色鸢尾
* 2：维吉尼亚鸢尾

predict\_dataset = tf.convert\_to\_tensor([  
    [5.1, 3.3, 1.7, 0.5,],  
    [5.9, 3.0, 4.2, 1.5,],  
    [6.9, 3.1, 5.4, 2.1]  
])  
  
predictions = model(predict\_dataset)  
  
for i, logits in enumerate(predictions):  
  class\_idx = tf.argmax(logits).numpy()  
  p = tf.nn.softmax(logits)[class\_idx]  
  name = class\_names[class\_idx]  
  print("Example {} prediction: {} ({:4.1f}%)".format(i, name, 100\*p))

实施例0预测：Iris setosa（97.6％）

实施例1预测：虹膜鸢尾（90.8％）

实施例2预测：Iris virginica（71.1％）

使用Estimators构建线性模型

本教程使用TensorFlow中的[tf.estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator) API来解决基准二元分类问题。Estimator是TensorFlow最具扩展性并且面向生产的TensorFlow模型类型。有关更多信息，请参阅[Estimator指南](https://tensorflow.google.cn/guide/estimators)。

概述

我们使用包含人的年龄，教育程度，婚姻状况和职业（*特征*）数据的人口普查数据，尝试预测该人每年的收入是否超过5万美元（目标*标签*）。我们将训练一个*逻辑回归*模型，根据个人的信息，这个模型会输出0到1之间的数字可将该值解读为个人年收入超过 5 万美元的概率。

**关键点：**作为建模人员和开发人员，需要考虑如何使用此数据以及模型预测可能带来的潜在好处和危害。像这样的模型可能会加剧社会偏见和差异。每个特征是否与您要解决的问题相关，或者是否会引入偏见？有关更多信息，请阅读[ML公平性](https://developers.google.cn/machine-learning/fairness-overview/)。

设置

导入TensorFlow，特征列支持和支持模块：

import tensorflow as tf  
import tensorflow.feature\_column as fc   
  
import os  
import sys  
  
import matplotlib.pyplot as plt  
from IPython.display import clear\_output

我们在执行它时启用 [Eager Execution](https://tensorflow.google.cn/guide/eager)来检查这个程序：

tf.enable\_eager\_execution()

下载官方模型

我们将使用TensorFlow [模型库中](https://github.com/tensorflow/models/)提供的[宽度模型和深度模型](https://github.com/tensorflow/models/tree/master/official/wide_deep/)。下载代码，将根目录添加到Python路径，然后跳转到wide\_deep目录：

! pip install -q requests  
! git clone --depth 1 https://github.com/tensorflow/models

您使用的是pip版本18.0，但版本18.1可用。

您应该考虑通过'pip install --upgrade pip'命令进行升级。

克隆成'模特'......

remote：枚举对象：2978，完成。

remote：计算对象：100％（2978/2978），完成。

remote：压缩对象：100％（2578/2578），完成。

远程：总计2978（delta 507），重复使用1794（delta 324），打包重用0

接收物体：100％（2978/2978），376.92 MiB | 完成19.63 MiB / s。

解决增量：100％（507/507），完成。

检查连接...完成。

将存储库的根目录添加到Python路径：

models\_path = os.path.join(os.getcwd(), 'models')  
  
sys.path.append(models\_path)

下载数据集：

from official.wide\_deep import census\_dataset  
from official.wide\_deep import census\_main  
  
census\_dataset.download("/tmp/census\_data/")

命令行用法

该代码库包含一个完整的程序，可用于对此类模型进行实验。

要从命令行执行教程代码，首先将tensorflow / models的路径添加到您的PYTHONPATH。

#export PYTHONPATH=${PYTHONPATH}:"$(pwd)/models"  
#running from python you need to set the `os.environ` or the subprocess will not see the directory.  
  
if "PYTHONPATH" in os.environ:  
  os.environ['PYTHONPATH'] += os.pathsep +  models\_path  
else:  
  os.environ['PYTHONPATH'] = models\_path

使用--help，看看有什么命令行选项：

!python -m official.wide\_deep.census\_main --help

2018-10-09 16：17：02.716687：I tensorflow / core / platform / cpu\_feature\_guard.cc：141]您的CPU支持未编译此TensorFlow二进制文件的指令：AVX2 FMA

在人口普查收入数据集上培训DNN。

标志：

/docker/output/models/official/wide\_deep/census\_main.py：

-bs， - 的batch\_size：

用于培训和评估的批量大小。当使用多个gpus时，这是

该

所有设备的全局批量大小。例如，如果批次大小为32和

有4个GPU，每个GPU将在每一步获得8个示例。

（默认：'40'）

（整数）

- [否]清洁：

如果设置，则将删除model\_dir（如果存在）。

（默认：'false'）

-dd， - DATA\_DIR：

输入数据的位置。

（默认：'/ tmp / census\_data'）

- [无] download\_if\_missing：

如果数据尚未存在，则将数据下载到data\_dir。

（默认：'true'）

-ebe， - epochs\_between\_evals：

评估之间运行的训练时期的数量。

（默认：'2'）

（整数）

-ed， - export\_dir：

如果设置，模型的SavedModel序列化将导出到此

培训结束时的目录。有关更多详细信息，请参阅自述文件

相应

链接。

-hk， - 钩：

用于指定训练挂钩名称的（不区分大小写）字符串的列表。

﻿ 钩子：

﻿ loggingmetrichook

﻿ loggingtensorhook

﻿ examplespersecondhook

﻿ profilerhook

﻿ 示例：`--hooks ProfilerHook，ExamplesPerSecondHook`

有关详细信息，请参阅official.utils.logs.hooks\_helper。

（默认：'LoggingTensorHook'）

（以逗号分隔的列表）

-MD， - model\_dir：

模型检查点文件的位置。

（默认：'/ tmp / census\_model'）

-mt， - model\_type：<wide | deep | wide\_deep>：选择模型拓扑。

（默认：'wide\_deep'）

- 叔 - train\_epochs：

用于训练的时代数。

（默认：'40'）

（整数）

尝试--helpfull获取所有标志的列表。

现在运行模型：

!python -m official.wide\_deep.census\_main --model\_type=wide --train\_epochs=2

2018-10-09 16：17：05.048463：I tensorflow / core / platform / cpu\_feature\_guard.cc：141]您的CPU支持未编译此TensorFlow二进制文件的指令：AVX2 FMA

I1009 16：17：05.053292 140061072475904 tf\_logging.py:115]使用config：{'\_ num\_ps\_replicas'：0，'\_ eval\_distribute'：无，'\_ log\_step\_count\_steps'：100，'\_ service'：无，'\_ session\_config'：device\_count {

关键：“GPU”

}

，'\_ device\_fn'：无，'\_ train\_distribute'：无，'\_ prototocol'：无，'\_ save\_summary\_steps'：100，'\_ master'：''，'\_ evaluation\_master'：''，'\_ model\_dir'：'/ tmp / census\_model'， '\_num\_worker\_replicas'：1，'\_ cluster\_pec'：<tensorflow.python.training.server\_lib.ClusterSpec对象位于0x7f622d45bba8>，'\_ task\_id'：0，'\_ tf\_random\_seed'：无，'\_ global\_id\_in\_cluster'：0，'\_ save\_checkpoints\_secs'：600，' \_save\_checkpoints\_steps'：无，'\_ kep\_checkpoint\_max'：5，'\_ is\_chief'：是的，'\_ keep\_checkpoint\_every\_n\_hours'：10000，'\_ expperimental\_distribute'：无，'\_ task\_type'：'worker'}

W1009 16：17：05.054460 140061072475904 tf\_logging.py:120]'cpuinfo'未导入。不会记录CPU信息。

W1009 16：17：05.054889 140061072475904 tf\_logging.py:120]'psutil'未导入。内存信息不会被记录。

I1009 16：17：05.192945 140061072475904 tf\_logging.py:115]基准测试：{'run\_parameters'：[{'long\_value'：40，'name'：'batch\_size'}，{'string\_value'：'wide'，'name' ：'model\_type'}，{'long\_value'：2，'name'：'train\_epochs'}]，'run\_date'：'2018-10-09T16：17：05.053988Z'，'model\_name'：'wide\_deep'，'test\_id '：无，'tensorflow\_version'：{'git\_hash'：“b'unknown'”，'版本'：'1.11.0'}，'machine\_config'：{'gpu\_info'：{'count'：0}}，' tensorflow\_environment\_variables'：[]，'dataset'：{'name'：'Census Income'}}

I1009 16：17：05.237414 140061072475904 tf\_logging.py:115]解析/tmp/census\_data/adult.data

I1009 16：17：05.273519 140061072475904 tf\_logging.py:115]调用model\_fn。

I1009 16：17：06.173745 140061072475904 tf\_logging.py:115]完成调用model\_fn。

I1009 16：17：06.174018 140061072475904 tf\_logging.py:115]创建CheckpointSaverHook。

I1009 16：17：06.549936 140061072475904 tf\_logging.py:115]图表已完成。

I1009 16：17：06.652294 140061072475904 tf\_logging.py:115]运行local\_init\_op。

I1009 16：17：06.678956 140061072475904 tf\_logging.py:115]完成运行local\_init\_op。

I1009 16：17：07.380616 140061072475904 tf\_logging.py:115]将检查点0保存到/tmp/census\_model/model.ckpt中。

I1009 16：17：07.949594 140061072475904 tf\_logging.py:115] average\_loss = 0.6931472，loss = 27.725887

I1009 16：17：07.950152 140061072475904 tf\_logging.py:115] loss = 27.725887，step = 1

I1009 16：17：08.538611 140061072475904 tf\_logging.py:115] global\_step / sec：169.622

I1009 16：17：08.539464 140061072475904 tf\_logging.py:115] average\_loss = 0.39386895，损失= 15.754758（0.590秒）

I1009 16：17：08.539777 140061072475904 tf\_logging.py:115]损失= 15.754758，步长= 101（0.590秒）

I1009 16：17：08.878756 140061072475904 tf\_logging.py:115] global\_step / sec：293.969

I1009 16：17：08.879661 140061072475904 tf\_logging.py:115] average\_loss = 0.34996048，损失= 13.998419（0.340秒）

I1009 16：17：08.879964 140061072475904 tf\_logging.py:115]损失= 13.998419，步长= 201（0.340秒）

I1009 16：17：09.214598 140061072475904 tf\_logging.py:115] global\_step / sec：297.757

I1009 16：17：09.215559 140061072475904 tf\_logging.py:115] average\_loss = 0.23998952，损失= 9.599581（0.336秒）

I1009 16：17：09.215908 140061072475904 tf\_logging.py:115]损失= 9.599581，步骤= 301（0.336秒）

I1009 16：17：09.553291 140061072475904 tf\_logging.py:115] global\_step / sec：295.249

I1009 16：17：09.554064 140061072475904 tf\_logging.py:115] average\_loss = 0.3641164，loss = 14.564656（0.339 sec）

I1009 16：17：09.554347 140061072475904 tf\_logging.py:115]损失= 14.564656，步骤= 401（0.338秒）

I1009 16：17：09.888134 140061072475904 tf\_logging.py:115] global\_step / sec：298.656

I1009 16：17：09.888899 140061072475904 tf\_logging.py:115] average\_loss = 0.37377656，损失= 14.951062（0.335秒）

I1009 16：17：09.889189 140061072475904 tf\_logging.py:115]损失= 14.951062，步骤= 501（0.335秒）

I1009 16：17：10.230372 140061072475904 tf\_logging.py:115] global\_step / sec：292.182

I1009 16：17：10.231162 140061072475904 tf\_logging.py:115] average\_loss = 0.44428167，损失= 17.771267（0.342秒）

I1009 16：17：10.231468 140061072475904 tf\_logging.py:115]损失= 17.771267，步骤= 601（0.342秒）

I1009 16：17：10.540647 140061072475904 tf\_logging.py:115] global\_step / sec：322.283

I1009 16：17：10.541343 140061072475904 tf\_logging.py:115] average\_loss = 0.35957533，loss = 14.383014（0.310 sec）

I1009 16：17：10.541604 140061072475904 tf\_logging.py:115]损失= 14.383014，步骤= 701（0.310秒）

I1009 16：17：10.869942 140061072475904 tf\_logging.py:115] global\_step / sec：303.721

I1009 16：17：10.870860 140061072475904 tf\_logging.py:115] average\_loss = 0.34024596，损失= 13.6098385（0.330秒）

I1009 16:17:10.871265 140061072475904 tf\_logging.py:115]损失= 13.6098385，步骤= 801（0.330秒）

I1009 16：17：11.255781 140061072475904 tf\_logging.py:115] global\_step / sec：259.158

I1009 16：17：11.256601 140061072475904 tf\_logging.py:115] average\_loss = 0.41650906，损失= 16.660362（0.386秒）

I1009 16：17：11.256903 140061072475904 tf\_logging.py:115]损失= 16.660362，步骤= 901（0.386秒）

I1009 16：17：11.594015 140061072475904 tf\_logging.py:115] global\_step / sec：295.654

I1009 16：17：11.594866 140061072475904 tf\_logging.py:115] average\_loss = 0.332558，损失= 13.3023205（0.338秒）

I1009 16：17：11.595208 140061072475904 tf\_logging.py:115]损失= 13.3023205，步长= 1001（0.338秒）

I1009 16：17：11.937734 140061072475904 tf\_logging.py:115] global\_step / sec：290.926

I1009 16：17：11.938567 140061072475904 tf\_logging.py:115] average\_loss = 0.2967804，损失= 11.871216（0.344秒）

I1009 16：17：11.938887 140061072475904 tf\_logging.py:115]损失= 11.871216，步长= 1101（0.344秒）

I1009 16：17：12.280265 140061072475904 tf\_logging.py:115] global\_step / sec：291.968

I1009 16：17：12.281141 140061072475904 tf\_logging.py:115] average\_loss = 0.40725145，loss = 16.290058（0.343 sec）

I1009 16：17：12.281456 140061072475904 tf\_logging.py:115]损失= 16.290058，步长= 1201（0.343秒）

I1009 16：17：12.610941 140061072475904 tf\_logging.py:115] global\_step / sec：302.411

I1009 16：17：12.611870 140061072475904 tf\_logging.py:115] average\_loss = 0.29060557，loss = 11.624223（0.331 sec）

I1009 16：17：12.612231 140061072475904 tf\_logging.py:115]损失= 11.624223，步长= 1301（0.331秒）

I1009 16：17：12.954074 140061072475904 tf\_logging.py:115] global\_step / sec：291.414

I1009 16：17：12.954836 140061072475904 tf\_logging.py:115] average\_loss = 0.4619999，损失= 18.479996（0.343秒）

I1009 16：17：12.955148 140061072475904 tf\_logging.py:115] loss = 18.479996，step = 1401（0.343 sec）

I1009 16：17：13.289181 140061072475904 tf\_logging.py:115] global\_step / sec：298.435

I1009 16：17：13.290104 140061072475904 tf\_logging.py:115] average\_loss = 0.34678543，loss = 13.871417（0.335 sec）

I1009 16：17：13.290464 140061072475904 tf\_logging.py:115]损失= 13.871417，步长= 1501（0.335秒）

I1009 16：17：13.629909 140061072475904 tf\_logging.py:115] global\_step / sec：293.484

I1009 16：17：13.630782 140061072475904 tf\_logging.py:115] average\_loss = 0.2735587，损失= 10.9423485（0.341秒）

I1009 16：17：13.631147 140061072475904 tf\_logging.py:115]损失= 10.9423485，步骤= 1601（0.341秒）

I1009 16：17：13.723575 140061072475904 tf\_logging.py:115]将1629的检查点保存到/tmp/census\_model/model.ckpt中。

I1009 16：17：13.844036 140061072475904 tf\_logging.py:115]最后一步的损失：0.046886865。

I1009 16：17：13.854052 140061072475904 tf\_logging.py:115]解析/tmp/census\_data/adult.test

I1009 16：17：13.882764 140061072475904 tf\_logging.py:115]调用model\_fn。

W1009 16：17：14.954841 140061072475904 tf\_logging.py:125]已知梯形法则产生不正确的PR-AUC; 请切换到“careful\_interpolation”。

W1009 16：17：14.972438 140061072475904 tf\_logging.py:125]已知梯形法则产生不正确的PR-AUC; 请切换到“careful\_interpolation”。

I1009 16：17：14.991327 140061072475904 tf\_logging.py:115]完成调用model\_fn。

I1009 16：17：15.009677 140061072475904 tf\_logging.py:115]在2018-10-09-16：17：15开始评估

I1009 16：17：15.126896 140061072475904 tf\_logging.py:115]图表已完成。

I1009 16：17：15.128459 140061072475904 tf\_logging.py:115]从/tmp/census\_model/model.ckpt-1629恢复参数

I1009 16：17：15.175018 140061072475904 tf\_logging.py:115]运行local\_init\_op。

I1009 16：17：15.204138 140061072475904 tf\_logging.py:115]完成运行local\_init\_op。

I1009 16：17：17.024853 140061072475904 tf\_logging.py:115]完成评估2018-10-09-16：17：17

I1009 16：17：17.025108 140061072475904 tf\_logging.py:115]保存全局步骤1629的dict：accuracy = 0.83711076，accuracy\_baseline = 0.76377374，auc = 0.8841798，auc\_precision\_recall = 0.69626415，average\_loss = 0.3507863，global\_step = 1629，label / mean = 0.23622628，损失= 13.997921，精度= 0.6953534，预测/平均值= 0.23868126，召回= 0.5525221

I1009 16：17：17.249652 140061072475904 tf\_logging.py:115]为全局步骤1629保存'checkpoint\_path'摘要：/tmp/census\_model/model.ckpt-1629

I1009 16：17：17.250328 140061072475904 tf\_logging.py:115]在纪元2/2的结果

I1009 16：17：17.250420 140061072475904 tf\_logging.py:115] ------------------------------------- -----------------------

I1009 16：17：17.250502 140061072475904 tf\_logging.py:115]准确度：0.83711076

I1009 16：17：17.250572 140061072475904 tf\_logging.py:115] accuracy\_baseline：0.76377374

I1009 16：17：17.250638 140061072475904 tf\_logging.py:115] auc：0.8841798

I1009 16：17：17.250701 140061072475904 tf\_logging.py:115] auc\_precision\_recall：0.69626415

I1009 16：17：17.250767 140061072475904 tf\_logging.py:115] average\_loss：0.3507863

I1009 16：17：17.250836 140061072475904 tf\_logging.py:115] global\_step：1629

I1009 16：17：17.250901 140061072475904 tf\_logging.py:115] label / mean：0.23622628

I1009 16：17：17.250980 140061072475904 tf\_logging.py:115]损失：13.997921

I1009 16：17：17.251052 140061072475904 tf\_logging.py:115]精度：0.6953534

I1009 16：17：17.251115 140061072475904 tf\_logging.py:115]预测/平均值：0.23868126

I1009 16：17：17.251177 140061072475904 tf\_logging.py:115]召回：0.5525221

I1009 16：17：17.251313 140061072475904 tf\_logging.py:115]基准指标：{'单位'：无，'价值'：0.8371107578277588，'时间戳'：'2018-10-09T16：17：17.251276Z'，'global\_step'： 1629，'name'：'准确'，'额外'：[]}

I1009 16：17：17.251432 140061072475904 tf\_logging.py:115]基准指标：{'unit'：无，'值'：0.7637737393379211，'timestamp'：'2018-10-09T16：17：17.251410Z'，'global\_step'： 1629，'name'：'accuracy\_baseline'，'extras'：[]}

I1009 16：17：17.251533 140061072475904 tf\_logging.py:115]基准指标：{'单位'：无，'价值'：0.8841797709465027，'时间戳'：'2018-10-09T16：17：17.251514Z'，'global\_step'： 1629，'name'：'auc'，'extras'：[]}

I1009 16：17：17.251632 140061072475904 tf\_logging.py:115]基准指标：{'unit'：无，'值'：0.6962641477584839，'timestamp'：'2018-10-09T16：17：17.251614Z'，'global\_step'： 1629，'name'：'auc\_precision\_recall'，'extras'：[]}

I1009 16：17：17.251730 140061072475904 tf\_logging.py:115]基准指标：{'单位'：无，'价值'：0.3507862985134125，'时间戳'：'2018-10-09T16：17：17.251711Z'，'global\_step'： 1629，'name'：'average\_loss'，'extras'：[]}

I1009 16：17：17.251827 140061072475904 tf\_logging.py:115]基准指标：{'unit'：无，'值'：0.23622627556324005，'timestamp'：'2018-10-09T16：17：17.251809Z'，'global\_step'： 1629，'name'：'label / mean'，'extras'：[]}

I1009 16：17：17.251924 140061072475904 tf\_logging.py:115]基准指标：{'unit'：无，'值'：13.997920989990234，'timestamp'：'2018-10-09T16：17：17.251905Z'，'global\_step'： 1629，'name'：'loss'，'extras'：[]}

I1009 16：17：17.252021 140061072475904 tf\_logging.py:115]基准指标：{'单位'：无，'价值'：0.6953533887863159，'时间戳'：'2018-10-09T16：17：17.252002Z'，'global\_step'： 1629，'name'：'precision'，'extras'：[]}

I1009 16：17：17.252116 140061072475904 tf\_logging.py:115]基准指标：{'unit'：无，'值'：0.23868125677108765，'timestamp'：'2018-10-09T16：17：17.252098Z'，'global\_step'： 1629，'name'：'预测/平均'，'额外'：[]}

I1009 16：17：17.252212 140061072475904 tf\_logging.py:115]基准指标：{'单位'：无，'价值'：0.5525221228599548，'时间戳'：'2018-10-09T16：17：17.252194Z'，'global\_step'： 1629，'名字'：'召回'，'额外'：[]}

读取美国人口普查数据

此示例使用1994年和1995年的[美国人口普查收入数据集](https://archive.ics.uci.edu/ml/datasets/Census+Income)。我们提供了[census\_dataset.py](https://github.com/tensorflow/models/tree/master/official/wide_deep/census_dataset.py)脚本来下载数据并执行一些数据清理工作。

由于任务是*二元分类问题*，我们将构造一个名为“label”的标签列，如果收入超过5万，则其值为1，否则为0。有关参考，请input\_fn在[census\_main.py](https://github.com/tensorflow/models/tree/master/official/wide_deep/census_main.py)。

让我们看看数据，看看我们可以用哪些列来预测目标标签：

!ls  /tmp/census\_data/

adult.data adult.test

train\_file = "/tmp/census\_data/adult.data"  
test\_file = "/tmp/census\_data/adult.test"

[pandas](https://pandas.pydata.org/)为数据分析提供了一些方便的实用程序。以下是收入普查数据集中可用列的列表：

import pandas  
  
train\_df = pandas.read\_csv(train\_file, header = None, names = census\_dataset.\_CSV\_COLUMNS)  
test\_df = pandas.read\_csv(test\_file, header = None, names = census\_dataset.\_CSV\_COLUMNS)  
  
train\_df.head()

|  | 年龄 | workclass | fnlwgt | 教育 | education\_num | 婚姻状况 | 占用 | 关系 | 种族 | 性别 | 资本收益 | capital\_loss | 每周几小时 | 祖国 | income\_bracket |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 39 | 国家政务 | 77516 | 学士 | 13 | 从未结婚 | ADM-文书 | 未在家庭 | 白色 | 男 | 2174 | 0 | 40 | 美国 - 美国 | <= 50K |
| 1 | 50 | 自EMP-不-INC | 83311 | 学士 | 13 | 已婚，CIV配偶 | EXEC-管理 | 丈夫 | 白色 | 男 | 0 | 0 | 13 | 美国 - 美国 | <= 50K |
| 2 | 38 | 私人的 | 215646 | HS-毕业生 | 9 | 离婚 | 处理程序，清洁剂 | 未在家庭 | 白色 | 男 | 0 | 0 | 40 | 美国 - 美国 | <= 50K |
| 3 | 53 | 私人的 | 234721 | 第11 | 7 | 已婚，CIV配偶 | 处理程序，清洁剂 | 丈夫 | 黑色 | 男 | 0 | 0 | 40 | 美国 - 美国 | <= 50K |
| 4 | 28 | 私人的 | 338409 | 学士 | 13 | 已婚，CIV配偶 | 教授专科 | 妻子 | 黑色 | 女 | 0 | 0 | 40 | 古巴 | <= 50K |

列分为两种类型：*类别*列和*连续*列：

* 如果列的值只能是有限集中的一个*类别，*则该列称为*类别列*。例如，婚恋状况（妻子、丈夫、未婚等）或受教育程度（高中、大学等）属于类别列。
* 如果列的值可以是连续范围内的任何数值，则该列称为*连续列*。例如，一个人的资本收益（例如$ 14,084）是一个连续的列。

将数据转换为张量

构建[tf.estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator)模型时，使用*输入函数*（或input\_fn）指定输入数据。此构建器函数返回批量(features-dict, label)对的[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)。在传递给 tf.estimator.Estimator 方法（如 train 和 evaluate）之前不会进行调用。

输入构建器函数返回以下对：

1. features：从特征名称到包含批量特征的Tensors或SparseTensors的映射字典。
2. labels：包含批次标签的Tensor.

其中的键features用于配置模型的输入层。

**注意：**在构造TensorFlow图时调用输入函数，而*不是*在运行图时调用。该调用会返回输入数据的表示法（表示为一系列 TensorFlow 图操作）。

对于像这样的小问题，可以通过拆分[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)来构建pandas.DataFrame：

def easy\_input\_function(df, label\_key, num\_epochs, shuffle, batch\_size):  
  label = df[label\_key]  
  ds = tf.data.Dataset.from\_tensor\_slices((dict(df),label))  
  
  if shuffle:  
    ds = ds.shuffle(10000)  
  
  ds = ds.batch(batch\_size).repeat(num\_epochs)  
  
  return ds

由于我们已启用了 Eager Execution，因此可以轻松检查生成的数据集：

ds = easy\_input\_function(train\_df, label\_key='income\_bracket', num\_epochs=5, shuffle=True, batch\_size=10)  
  
for feature\_batch, label\_batch in ds.take(1):  
  print('Some feature keys:', list(feature\_batch.keys())[:5])  
  print()  
  print('A batch of Ages  :', feature\_batch['age'])  
  print()  
  print('A batch of Labels:', label\_batch )

一些功能键：['hours\_per\_week'，'native\_country'，'capital\_loss'，'education\_num'，'职业']

一批年龄：tf.Tensor（[19 43 45 25 35 24 21 34 32 37]，shape =（10，），dtype = int32）

一批标签：tf.Tensor（

[b'<= 50K'b'<= 50K'b'> 50K'b'<= 50K'b'<= 50K'b'<= 50K'b'<= 50K'b'<= 50K'

b'> 50K'b'<= 50K']，shape =（10，），dtype = string）

但是这种方法的可扩展性很差。更大的数据集应该从磁盘流式传输。census\_dataset.input\_fn提供了如何使用 tf.decode\_csv 和 tf.data.TextLineDataset 执行这种操作的示例

import inspect  
print(inspect.getsource(census\_dataset.input\_fn))

def input\_fn（data\_file，num\_epochs，shuffle，batch\_size）：

msgstr“”“为Estimator生成输入函数。”“

断言tf.gfile.Exists（data\_file），（

'％s未找到。请确保您已运行census\_dataset.py和'

'将--data\_dir参数设置为正确的路径。％data\_file）

def parse\_csv（value）：

tf.logging.info（'Parsing {}'。format（data\_file））

columns = tf.decode\_csv（value，record\_defaults = \_CSV\_COLUMN\_DEFAULTS）

features = dict（zip（\_CSV\_COLUMNS，columns））

labels = features.pop（'income\_bracket'）

classes = tf.equal（标签，'> 50K'）＃二进制分类

返回功能，类

＃使用Dataset API从输入文件中提取行。

dataset = tf.data.TextLineDataset（data\_file）

如果洗牌：

dataset = dataset.shuffle（buffer\_size = \_NUM\_EXAMPLES ['train']）

dataset = dataset.map（parse\_csv，num\_parallel\_calls = 5）

＃我们在洗牌之后调用重复，而不是之前，以防止分开

#epochs混合在一起。

dataset = dataset.repeat（num\_epochs）

dataset = dataset.batch（batch\_size）

返回数据集

input\_fn将返回等效输出：

ds = census\_dataset.input\_fn(train\_file, num\_epochs=5, shuffle=True, batch\_size=10)  
  
for feature\_batch, label\_batch in ds.take(1):  
  print('Feature keys:', list(feature\_batch.keys())[:5])  
  print()  
  print('Age batch   :', feature\_batch['age'])  
  print()  
  print('Label batch :', label\_batch )

信息：tensorflow：解析/tmp/census\_data/adult.data

警告：在标记解析之前记录到stderr。

I1009 16：17：19.018875 140156615698176 tf\_logging.py:115]解析/tmp/census\_data/adult.data

功能键：['hours\_per\_week'，'native\_country'，'capital\_loss'，'education\_num'，'职业']

年龄批次：tf.Tensor（[37 42 27 19 29 59 46 54 52 35]，shape =（10，），dtype = int32）

标签批量：tf.Tensor（[假错假错假错假错真假]，形状=（10，），dtype = bool）

因为Estimators期望input\_fn不带参数，我们通常将可配置的输入函数包装成具有预期签名的obejct。对于这个笔记本配置train\_inpf，迭代数据两次：

import functools  
  
train\_inpf = functools.partial(census\_dataset.input\_fn, train\_file, num\_epochs=2, shuffle=True, batch\_size=64)  
test\_inpf = functools.partial(census\_dataset.input\_fn, test\_file, num\_epochs=1, shuffle=False, batch\_size=64)

为模型选择特征并进行特征工程特征

Estimator使用称为[特征列](https://tensorflow.google.cn/guide/feature_columns)的机制来描述模型如何解释每个原始输入特征。Estimator需要数值输入向量，而特征列会描述模型应如何转换每个特征。

选择和创建一组正确的特征列是学习有效模型的关键。*特征列*可以是在原始的特征dict中的任一个原始输入（*基准特征柱*），或使用在一个或多个基准转换而创建的任意新列（衍生*特征列*）。

特征列是可用于预测目标标签的任何原始或衍生变量的抽象概念。

基准功能列

数值列

最简单的feature\_column是numeric\_column。它表示特征是直接输入到模型中的数值。例如：

age = fc.numeric\_column('age')

模型将使用这些feature\_column定义来构建模型输入。您可以使用input\_layer函数检查生成结果的输出：

fc.input\_layer(feature\_batch, [age]).numpy()

阵列（[[37]，

[42]，

[27]，

[19]，

[29]，

[59.]，

[46]，

[54]，

[52]，

[35.]]，dtype = float32）

以下将仅使用age特征训练和评估模型：

classifier = tf.estimator.LinearClassifier(feature\_columns=[age])  
classifier.train(train\_inpf)  
result = classifier.evaluate(test\_inpf)  
  
clear\_output()  # used for display in notebook  
print(result)

{'auc'：0.6781993，'auc\_precision\_recall'：0.31136066，'average\_loss'：0.52740055，'precision'：0.76377374，'label / mean'：0.23622628，'recall'：0.0，'precision'：0.0，'accuracy\_baseline'：0.76377374 ，'global\_step'：1018，'预测/均值'：0.20646697，'亏损'：33.672974}

同样，我们可以为模型中使用的每个连续特征列定义NumericColumn：

education\_num = tf.feature\_column.numeric\_column('education\_num')  
capital\_gain = tf.feature\_column.numeric\_column('capital\_gain')  
capital\_loss = tf.feature\_column.numeric\_column('capital\_loss')  
hours\_per\_week = tf.feature\_column.numeric\_column('hours\_per\_week')  
  
my\_numeric\_columns = [age,education\_num, capital\_gain, capital\_loss, hours\_per\_week]  
  
fc.input\_layer(feature\_batch, my\_numeric\_columns).numpy()

数组（[[37。，0.，0.，16.，35。]，

[42.，0.，0.，9.，40。]，

[27.，0.，0.，13。，50。]，

[19.，0.，0.，9.，30。]，

[29.，0.，0.，13.，40。]，

[59。，0.，0.，9.，35。]，

[46.，0.，0.，15.，48。]，

[54。，0.，0.，6.，60。]，

[52.，0.，0.，9.，52。]，

[35.，0.，0.，9.，40。]]，dtype = float32）

您可以通过更改构造函数feature\_columns的参数来重新训练这些功能的模型：

classifier = tf.estimator.LinearClassifier(feature\_columns=my\_numeric\_columns)  
classifier.train(train\_inpf)  
  
result = classifier.evaluate(test\_inpf)  
  
clear\_output()  
  
for key,value in sorted(result.items()):  
  print('%s: %s' % (key, value))

准确度：0.78250724

accuracy\_baseline：0.76377374

auc：0.70410156

auc\_precision\_recall：0.49718997

average\_loss：2.7589335

global\_step：1018

label / mean：0.23622628

损失：176.14978

精度：0.61526835

预测/平均值：0.26403713

召回：0.21164846

类别列

要为类别列特征定义特征列，请使用其中一个tf.feature\_column.categorical\_column\*函数创建CategoricalColumn。

如果您知道列的所有可能特征值的集合 - 并且集合中只有少数特征值 – 请使用categorical\_column\_with\_vocabulary\_list。列表中的每个键都被分配一个自动递增的ID（从0开始）。例如，对于relationship列，我们可以将整数ID0分配给特征字符串Husband，将1分配给 “Not-in-family”，以此类推。

relationship = fc.categorical\_column\_with\_vocabulary\_list(  
    'relationship',  
    ['Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried', 'Other-relative'])

这将从原始输入要素创建稀疏的独热向量。

我们使用的input\_layer函数是为DNN模型设计的，要求提供密集输入。为了演示类别列，我们必须将其封装在[tf.feature\_column.indicator\_column](https://tensorflow.google.cn/api_docs/python/tf/feature_column/indicator_column)以创建密集的独热输出（线性Estimators通常可以跳过此密集步骤）。

**注意：**还有一种稀疏到密集的函数是[**tf.feature\_column.embedding\_column**](https://tensorflow.google.cn/api_docs/python/tf/feature_column/embedding_column)。

运行使用age和relationship列进行配置的输入层：

fc.input\_layer(feature\_batch, [age, fc.indicator\_column(relationship)])

如果我们事先不知道可能值集，请使用categorical\_column\_with\_hash\_bucket：

occupation = tf.feature\_column.categorical\_column\_with\_hash\_bucket(  
    'occupation', hash\_bucket\_size=1000)

在这里，我们在训练中看到的特征列occupation中的每个可能值都已经过哈希处理变为整数ID。以下示例包含几个不同的职业：

for item in feature\_batch['occupation'].numpy():  
    print(item.decode())

教授专科

机运inspct

教授专科

教授专科

教授专科

农牧渔业

农牧渔业

机运inspct

保护-SERV

？

如果我们使用哈希处理的列运行input\_layer，我们会看到输出形状是(batch\_size, hash\_bucket\_size)：

occupation\_result = fc.input\_layer(feature\_batch, [fc.indicator\_column(occupation)])  
  
occupation\_result.numpy().shape

（10,1000）

如果我们对hash\_bucket\_size维度执行[tf.argmax](https://tensorflow.google.cn/api_docs/python/tf/argmax)操作，则更容易看到实际结果。请注意，任何重复的职业都映射到相同的伪随机索引：

tf.argmax(occupation\_result, axis=1).numpy()

阵列（[979,911,979,979,979,936,936,911,684,65]）

**注意：**哈希冲突是不可避免的，但通常对模型质量的影响很小。如果散列桶用于压缩输入空间，则效果可能是显着的。有关这些哈希冲突影响的更直观的示例，请参阅[此笔记本](https://colab.research.google.com/github/tensorflow/models/blob/master/samples/outreach/blogs/housing_prices.ipynb)。

无论我们如何选择定义 SparseColumn，都可以通过查找固定映射或哈希处理将各个特征字符串映射到整数ID。事实上，LinearModel类负责管理映射并创建[tf.Variable](https://tensorflow.google.cn/api_docs/python/tf/Variable)来存储每个特征ID 的模型参数（模型*权重*）。这些模型参数是通过稍后介绍的模型训练过程来学习的。

我们可以用类似的技巧来定义其他分类特征：

education = tf.feature\_column.categorical\_column\_with\_vocabulary\_list(  
    'education', [  
        'Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',  
        'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',  
        '5th-6th', '10th', '1st-4th', 'Preschool', '12th'])  
  
marital\_status = tf.feature\_column.categorical\_column\_with\_vocabulary\_list(  
    'marital\_status', [  
        'Married-civ-spouse', 'Divorced', 'Married-spouse-absent',  
        'Never-married', 'Separated', 'Married-AF-spouse', 'Widowed'])  
  
workclass = tf.feature\_column.categorical\_column\_with\_vocabulary\_list(  
    'workclass', [  
        'Self-emp-not-inc', 'Private', 'State-gov', 'Federal-gov',  
        'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'])  
  
  
my\_categorical\_columns = [relationship, occupation, education, marital\_status, workclass]

使用两组列可以轻松配置使用所有这些特征的模型：

classifier = tf.estimator.LinearClassifier(feature\_columns=my\_numeric\_columns+my\_categorical\_columns)  
classifier.train(train\_inpf)  
result = classifier.evaluate(test\_inpf)  
  
clear\_output()  
  
for key,value in sorted(result.items()):  
  print('%s: %s' % (key, value))

准确度：0.79823107

accuracy\_baseline：0.76377374

auc：0.6876817

auc\_precision\_recall：0.50509375

average\_loss：3.3195345

global\_step：1018

label / mean：0.23622628

损失：211.94252

精度：0.6521975

预测/平均值：0.14528999

召回：0.3125325

衍生特征列

通过分桶将连续特征变成类别特征。

有时，连续特征与标签之间的关系不是线性的。例如，*年龄*和*收入* - 一个人的收入可能在其职业生涯的早期阶段增长，然后在某一阶段时候增长放缓，最后，在退休后减少。在这种情况下，使用原始 age作为实值特征列可能不是一个好的选择，因为模型只能学习以下三种情况之一：

1. 随着年龄的增长，收入总是以某种速度增长（正相关），
2. 随着年龄的增长，收入总是以某种速度下降（负相关），或
3. 无论在什么年龄（无相关性），收入都保持不变。

如果我们想分别了解收入和每个年龄组之间的精细关系，我们可以利用*分桶技巧*。分桶是将连续特征的整个范围划分为一组连续桶，然后根据该值落入哪个桶将原始数字特征转换为桶ID（作为分类特征）的过程。因此，针对以上age，我们可以定义一个bucketized\_column为：

age\_buckets = tf.feature\_column.bucketized\_column(  
    age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])

boundaries是一个桶边界列表。在本例中，有10个边界，因此会生成11个年龄组桶（从17岁及以下，18-24岁，25-29岁，......，到65岁及以上）。

通过分组，模型将每个桶视为一个独热功能：

fc.input\_layer(feature\_batch, [age, age\_buckets]).numpy()

数组（[[37。，0.0,0.0,0.0,1.0,0.0,0.0,0.0,0。]，

[42.，0.0,0.0,0.0,0.1,0.0,0.0,0.0,0]，

[27.，0.0,0.1,0.0,0.0,0.0,0.0,0.0,0]，

[19.，0.，1.，0.，0.，0.，0.，0.，0.，0.，0.，0。]，

[29.，0.0,0.1,0.0,0.0,0.0,0.0,0.0,0。]，

[59。，0.，0.0,0.0,0.0,0.0,0.0,1.0,0。]，

[46.，0.0,0.0,0.0,0.0,0.1,0.0,0.0,0]，

[54.，0.0,0.0,0.0,0.0,0.0,1.0,0.0,0]，

[52.，0.，0.0,0.0,0.0,0.0,0.1,0.0,0。]，

[35.，0.0,0.0,0.1,1.0,0.0,0.0,0.0,0]，

D型= FLOAT32）

通过组合列学习复杂的关系

单独使用每个基准特征列可能不足以解释数据。例如，教育与标签（收入> 50,000美元）之间的相关性可能因职业的不同而不同。因此，如果我们只学了education="Bachelors"和education="Masters"的单个模型权重，无法获得每一个教育职业组合（例如，区分education="Bachelors"与occupation="Exec-managerial"和 education="Bachelors" 与occupation="Craft-repair"）。

要了解不同要素组合之间的差异，我们可以向模型添加组合特征列：

education\_x\_occupation = tf.feature\_column.crossed\_column(  
    ['education', 'occupation'], hash\_bucket\_size=1000)

我们还可以针对两个以上的列创建crossed\_column。每个组成列可以是基准特征列（SparseColumn）、分桶实值特征列，也可以是其他CrossColumn。例如：

age\_buckets\_x\_education\_x\_occupation = tf.feature\_column.crossed\_column(  
    [age\_buckets, 'education', 'occupation'], hash\_bucket\_size=1000)

这些组合列始终使用哈希分桶来避免类别数量的指数爆炸，并将控制数量的模型权重的权利置于用户手中。

对于可视化带有交叉列的散列桶的效果的实例，请参阅[这个教程](https://colab.research.google.com/github/tensorflow/models/blob/master/samples/outreach/blogs/housing_prices.ipynb)

定义逻辑回归模型

在处理输入数据并定义所有特征列之后，我们可以将它们组合在一起并构建*逻辑回归*模型。上一节介绍了几种基准特征列和衍生特征列，包括：

* CategoricalColumn
* NumericColumn
* BucketizedColumn
* CrossedColumn

所有这些都是抽象FeatureColumn类的子类，可以添加到模型的feature\_columns字段中：

import tempfile  
  
base\_columns = [  
    education, marital\_status, relationship, workclass, occupation,  
    age\_buckets,  
]  
  
crossed\_columns = [  
    tf.feature\_column.crossed\_column(  
        ['education', 'occupation'], hash\_bucket\_size=1000),  
    tf.feature\_column.crossed\_column(  
        [age\_buckets, 'education', 'occupation'], hash\_bucket\_size=1000),  
]  
  
model = tf.estimator.LinearClassifier(  
    model\_dir=tempfile.mkdtemp(),   
    feature\_columns=base\_columns + crossed\_columns,  
    optimizer=tf.train.FtrlOptimizer(learning\_rate=0.1))

信息：tensorflow：使用默认配置。

I1009 16：17：43.038667 140156615698176 tf\_logging.py:115]使用默认配置。

INFO:tensorflow:Using config: {'\_session\_config': allow\_soft\_placement: true

graph\_options {

rewrite\_options {

meta\_optimizer\_iterations: ONE

}

}

, '\_save\_checkpoints\_steps': None, '\_cluster\_spec': <tensorflow.python.training.server\_lib.ClusterSpec object at 0x7f7802f8a198>, '\_eval\_distribute': None, '\_save\_checkpoints\_secs': 600, '\_save\_summary\_steps': 100, '\_global\_id\_in\_cluster': 0, '\_service': None, '\_is\_chief': True, '\_tf\_random\_seed': None, '\_protocol': None, '\_model\_dir': '/tmp/tmpews\_wh40', '\_task\_id': 0, '\_device\_fn': None, '\_train\_distribute': None, '\_num\_worker\_replicas': 1, '\_keep\_checkpoint\_every\_n\_hours': 10000, '\_log\_step\_count\_steps': 100, '\_task\_type': 'worker', '\_num\_ps\_replicas': 0, '\_master': '', '\_experimental\_distribute': None, '\_keep\_checkpoint\_max': 5, '\_evaluation\_master': ''}

I1009 16：17：43.042024 140156615698176 tf\_logging.py:115]使用config：{'\_ session\_config'：allow\_soft\_placement：true

graph\_options {

rewrite\_options {

meta\_optimizer\_iterations：ONE

}

}

，'\_save\_checkpoints\_steps'：无，'\_ cluster\_pecs'：<tensorflow.python.training.server\_lib.ClusterSpec对象位于0x7f7802f8a198>，'\_ eval\_distribute'：无，'\_ \_ save\_checkpoints\_secs'：600，'\_ save\_summary\_steps'：100，'\_ global\_id\_in\_cluster'：0， '\_service'：无，'\_ is\_chief'：是的，'\_ tf\_random\_seed'：无，'\_ prototocol'：无，'\_ model\_dir'：'/ tmp / tmpews\_wh40'，'\_ task\_id'：0，'\_ device\_fn'：无，'\_ \_ train\_distribute' ：无，'\_ num\_worker\_replicas'：1，'\_ keep\_checkpoint\_every\_n\_hours'：10000，'\_ log\_step\_count\_steps'：100，'\_ task\_type'：'worker'，'\_ num\_ps\_replicas'：0，'\_ master'：''，'\_ expperimental\_distribute'：无，'\_ keep\_checkpoint\_max '：5，'\_evaluation\_master'：''}

该模型自动学习偏差项，偏差项可以控制预测而不需要观察任何特征。学习的模型文件存储在model\_dir。

训练和评估模型

将所有功能添加到模型后，我们开始训练模型。训练模型只需要使用一个[tf.estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator)API 命令：

train\_inpf = functools.partial(census\_dataset.input\_fn, train\_file,   
                               num\_epochs=40, shuffle=True, batch\_size=64)  
  
model.train(train\_inpf)  
  
clear\_output()  # used for notebook display

训练模型后，通过预测维持数据的标签来评估模型的准确性：

results = model.evaluate(test\_inpf)  
  
clear\_output()  
  
for key,value in sorted(result.items()):  
  print('%s: %0.2f' % (key, value))

准确度：0.80

accuracy\_baseline：0.76

澳柯：0.69

auc\_precision\_recall：0.51

average\_loss：3.32

global\_step：1018.00

label / mean：0.24

损失：211.94

精度：0.65

预测/平均值：0.15

回忆：0.31

输出的第一行应显示如下内容：accuracy: 0.83，这个是表示准确度为83％。您可以尝试使用更多功能和转换，看看您是否可以做得更好！

评估模型后，我们可以向模型输入某个人的信息，让其预测这个人的年收入是否超过50,000美元。

让我们更详细地了解模型的表现：

import numpy as np  
  
predict\_df = test\_df[:20].copy()  
  
pred\_iter = model.predict(  
    lambda:easy\_input\_function(predict\_df, label\_key='income\_bracket',  
                               num\_epochs=1, shuffle=False, batch\_size=10))  
  
classes = np.array(['<=50K', '>50K'])  
pred\_class\_id = []  
  
for pred\_dict in pred\_iter:  
  pred\_class\_id.append(pred\_dict['class\_ids'])  
  
predict\_df['predicted\_class'] = classes[np.array(pred\_class\_id)]  
predict\_df['correct'] = predict\_df['predicted\_class'] == predict\_df['income\_bracket']  
  
clear\_output()  
  
predict\_df[['income\_bracket','predicted\_class', 'correct']]

|  | income\_bracket | predicted\_class | 正确 |
| --- | --- | --- | --- |
| 0 | <= 50K | <= 50K | 真正 |
| 1 | <= 50K | <= 50K | 真正 |
| 2 | > 50K | <= 50K | 假 |
| 3 | > 50K | <= 50K | 假 |
| 4 | <= 50K | <= 50K | 真正 |
| 五 | <= 50K | <= 50K | 真正 |
| 6 | <= 50K | <= 50K | 真正 |
| 7 | > 50K | > 50K | 真正 |
| 8 | <= 50K | <= 50K | 真正 |
| 9 | <= 50K | <= 50K | 真正 |
| 10 | > 50K | <= 50K | 假 |
| 11 | <= 50K | > 50K | 假 |
| 12 | <= 50K | <= 50K | 真正 |
| 13 | <= 50K | <= 50K | 真正 |
| 14 | > 50K | <= 50K | 假 |
| 15 | > 50K | > 50K | 真正 |
| 16 | <= 50K | <= 50K | 真正 |
| 17 | <= 50K | <= 50K | 真正 |
| 18 | <= 50K | <= 50K | 真正 |
| 19 | > 50K | > 50K | 真正 |

有关端到端的示例，请下载我们的[示例代码](https://github.com/tensorflow/models/tree/master/official/wide_deep/census_main.py)并将model\_type标志设置为wide。

添加正则化以防止过度拟合

正则化是一种用于避免过度拟合的技术。当模型在训练的数据上表现良好，但在模型之前没有见过的测试数据上表现不佳的时候可能会发生过度拟合。当模型过于复杂时，例如相对于观察到的训练数据的数量具有太多参数时，就可能发生过度拟合。正则化可以控制模型的复杂性，使模型对于看不见的数据更加通用。

您可以使用以下代码将L1和L2正则化添加到模型中：

model\_l1 = tf.estimator.LinearClassifier(  
    feature\_columns=base\_columns + crossed\_columns,  
    optimizer=tf.train.FtrlOptimizer(  
        learning\_rate=0.1,  
        l1\_regularization\_strength=10.0,  
        l2\_regularization\_strength=0.0))  
  
model\_l1.train(train\_inpf)  
  
results = model\_l1.evaluate(test\_inpf)  
clear\_output()  
for key in sorted(results):  
  print('%s: %0.2f' % (key, results[key]))

准确度：0.84

accuracy\_baseline：0.76

奥克：0.88

auc\_precision\_recall：0.69

average\_loss：0.35

global\_step：20351.00

label / mean：0.24

损失：22.47

精度：0.69

预测/平均值：0.24

召回：0.55

model\_l2 = tf.estimator.LinearClassifier(  
    feature\_columns=base\_columns + crossed\_columns,  
    optimizer=tf.train.FtrlOptimizer(  
        learning\_rate=0.1,  
        l1\_regularization\_strength=0.0,  
        l2\_regularization\_strength=10.0))  
  
model\_l2.train(train\_inpf)  
  
results = model\_l2.evaluate(test\_inpf)  
clear\_output()  
for key in sorted(results):  
  print('%s: %0.2f' % (key, results[key]))

准确度：0.84

accuracy\_baseline：0.76

奥克：0.88

auc\_precision\_recall：0.69

average\_loss：0.35

global\_step：20351.00

label / mean：0.24

损失：22.46

精度：0.68

预测/平均值：0.24

回忆：0.57

这些正则化模型的表现并不比基本模型好很多。我们看一下模型的权重分布，以便更好地了解正则化的效果：

def get\_flat\_weights(model):  
  weight\_names = [  
      name for name in model.get\_variable\_names()  
      if "linear\_model" in name and "Ftrl" not in name]  
  
  weight\_values = [model.get\_variable\_value(name) for name in weight\_names]  
  
  weights\_flat = np.concatenate([item.flatten() for item in weight\_values], axis=0)  
  
  return weights\_flat  
  
weights\_flat = get\_flat\_weights(model)  
weights\_flat\_l1 = get\_flat\_weights(model\_l1)  
weights\_flat\_l2 = get\_flat\_weights(model\_l2)

模型具有许多由未使用的哈希分箱引起的零值权重（在某些列中，哈希分箱的数量多于类别数量）。我们可以在查看重量分布时屏蔽这些权重：

weight\_mask = weights\_flat != 0  
  
weights\_base = weights\_flat[weight\_mask]  
weights\_l1 = weights\_flat\_l1[weight\_mask]  
weights\_l2 = weights\_flat\_l2[weight\_mask]

现在绘制分布图：

plt.figure()  
\_ = plt.hist(weights\_base, bins=np.linspace(-3,3,30))  
plt.title('Base Model')  
plt.ylim([0,500])  
  
plt.figure()  
\_ = plt.hist(weights\_l1, bins=np.linspace(-3,3,30))  
plt.title('L1 - Regularization')  
plt.ylim([0,500])  
  
plt.figure()  
\_ = plt.hist(weights\_l2, bins=np.linspace(-3,3,30))  
plt.title('L2 - Regularization')  
\_=plt.ylim([0,500])

两种类型的正则化都将权重的分布向零靠近。L2正则化对分布的尾部具有更大的影响，消除了极端权重。L1正则化产生更精确的零值，在本例中它将~200设置为零。

# 如何使用TF-Hub构建简单的文本分类器

TF-Hub是一个共享机器学习专业知识的平台，包含了预先训练过的**模块**。本教程分为两个主要部分。

\*\* 简介：使用TF-Hub训练文本分类器\*\*

我们将使用TF-Hub文本嵌入模块来训练具有合理基准准确度的简单情感分类器。然后，我们将分析预测，以确保我们的模型是合理的，并提出改进措施以提高准确性。

\*\* 提高：转迁移分析\*\*

在本节中，我们将使用各种TF-Hub模块来比较它们对估算器准确性的影响，并展示迁移学习的优势和缺陷。

## 先行要求

* 对Tensorflow [预制估算框架的](https://tensorflow.google.cn/get_started/premade_estimators)基本知识。
* 熟悉Pandas库。

## 环境搭建

# Install the latest Tensorflow version.  
!pip install --quiet "tensorflow>=1.7"  
# Install TF-Hub.  
!pip install -q tensorflow-hub  
!pip install -q seaborn

更多有关安装Tensorflow的详细信息，[请](https://tensorflow.google.cn/install/)访问[https://www.tensorflow.org/install/](https://tensorflow.google.cn/install/)。

import tensorflow as tf  
import tensorflow\_hub as hub  
import matplotlib.pyplot as plt  
import numpy as np  
import os  
import pandas as pd  
import re  
import seaborn as sns

## 开始

## 数据

我们将尝试解决Mass等人的[大型电影评论数据集v1.0](http://ai.stanford.edu/~amaas/data/sentiment/)任务。数据集由IMDB电影评论组成，标记为1到10的积极性。任务是将评论标记为**否定**或**正面**。

# Load all files from a directory in a DataFrame.  
def load\_directory\_data(directory):  
  data = {}  
  data["sentence"] = []  
  data["sentiment"] = []  
  for file\_path in os.listdir(directory):  
    with tf.gfile.GFile(os.path.join(directory, file\_path), "r") as f:  
      data["sentence"].append(f.read())  
      data["sentiment"].append(re.match("\d+\_(\d+)\.txt", file\_path).group(1))  
  return pd.DataFrame.from\_dict(data)  
  
# Merge positive and negative examples, add a polarity column and shuffle.  
def load\_dataset(directory):  
  pos\_df = load\_directory\_data(os.path.join(directory, "pos"))  
  neg\_df = load\_directory\_data(os.path.join(directory, "neg"))  
  pos\_df["polarity"] = 1  
  neg\_df["polarity"] = 0  
  return pd.concat([pos\_df, neg\_df]).sample(frac=1).reset\_index(drop=True)  
  
# Download and process the dataset files.  
def download\_and\_load\_datasets(force\_download=False):  
  dataset = tf.keras.utils.get\_file(  
      fname="aclImdb.tar.gz",   
      origin="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb\_v1.tar.gz",   
      extract=True)  
    
  train\_df = load\_dataset(os.path.join(os.path.dirname(dataset),   
                                       "aclImdb", "train"))  
  test\_df = load\_dataset(os.path.join(os.path.dirname(dataset),   
                                      "aclImdb", "test"))  
    
  return train\_df, test\_df  
  
# Reduce logging output.  
tf.logging.set\_verbosity(tf.logging.ERROR)  
  
train\_df, test\_df = download\_and\_load\_datasets()  
train\_df.head()

从http://ai.stanford.edu/~amaas/data/sentiment/aclImdb\_v1.tar.gz下载数据

84131840/84125825 [==============================] - 9s 0us / step

|  | 句子 | 情绪 | 极性 |
| --- | --- | --- | --- |
| 0 | 由Johnny Weissmull主演的最后一部泰山电影...... | 2 | 0 |
| 1 | - 一个非常漂亮的红头发女人在等他...... | 7 | 1 |
| 2 | 好：<br /> <br />有效的配色方案。G... | 2 | 0 |
| 3 | 我在电影院看过“冰河世纪”而我... | 8 | 1 |
| 4 | 电影伟大的一个衡量标准我... | 10 | 1 |

## 模型

### 输入函数

[Estimator框架](https://tensorflow.google.cn/get_started/premade_estimators#overview_of_programming_with_estimators)提供了包装Pandas数据框架的[输入函数](https://tensorflow.google.cn/api_docs/python/tf/estimator/inputs/pandas_input_fn)。

# Training input on the whole training set with no limit on training epochs.  
train\_input\_fn = tf.estimator.inputs.pandas\_input\_fn(  
    train\_df, train\_df["polarity"], num\_epochs=None, shuffle=True)  
  
# Prediction on the whole training set.  
predict\_train\_input\_fn = tf.estimator.inputs.pandas\_input\_fn(  
    train\_df, train\_df["polarity"], shuffle=False)  
# Prediction on the test set.  
predict\_test\_input\_fn = tf.estimator.inputs.pandas\_input\_fn(  
    test\_df, test\_df["polarity"], shuffle=False)

### 特征列

TF-Hub提供了一个可以在给定的文本功能上应用模块，并进一步传递模块的输出[特征列](https://github.com/tensorflow/hub/blob/master/docs/api_docs/python/hub/text_embedding_column.md)。在本教程中，我们将使用[nnlm-en-dim128模块](https://tfhub.dev/google/nnlm-en-dim128/1)。本教程的主要目的是：

* 将**一批句子**以**1-D张量的字符串**作为输入该模块。
* 该模块负责**预处理句子**（例如删除标点符号和拆分空格）。
* 该模块适用于任何输入（例如，**nnlm-en-dim128**哈希词汇中没有的词汇）。

embedded\_text\_feature\_column = hub.text\_embedding\_column(  
    key="sentence",   
    module\_spec="https://tfhub.dev/google/nnlm-en-dim128/1")

### Estimator

我们可以使用[DNN分类器](https://tensorflow.google.cn/api_docs/python/tf/estimator/DNNClassifier) 进行分类（请注意，有关标签函数的不同建模方法将在本教程的末尾进行进一步说明）。

estimator = tf.estimator.DNNClassifier(  
    hidden\_units=[500, 100],  
    feature\_columns=[embedded\_text\_feature\_column],  
    n\_classes=2,  
    optimizer=tf.train.AdagradOptimizer(learning\_rate=0.003))

### 训练

训练Estimator以获得合理数量的执行步骤。

# Training for 1,000 steps means 128,000 training examples with the default  
# batch size. This is roughly equivalent to 5 epochs since the training dataset  
# contains 25,000 examples.  
estimator.train(input\_fn=train\_input\_fn, steps=1000);

## 预测

运行训练和测试集的进行预测。

train\_eval\_result = estimator.evaluate(input\_fn=predict\_train\_input\_fn)  
test\_eval\_result = estimator.evaluate(input\_fn=predict\_test\_input\_fn)  
  
print("Training set accuracy: {accuracy}".format(\*\*train\_eval\_result))  
print("Test set accuracy: {accuracy}".format(\*\*test\_eval\_result))

训练集精度：0.8025599718093872

测试集精度：0.7925599813461304

## 混淆矩阵

我们可以检查混淆矩阵来直观的查看错误分类的分布。

def get\_predictions(estimator, input\_fn):  
  return [x["class\_ids"][0] for x in estimator.predict(input\_fn=input\_fn)]  
  
LABELS = [  
    "negative", "positive"  
]  
  
# Create a confusion matrix on training data.  
with tf.Graph().as\_default():  
  cm = tf.confusion\_matrix(train\_df["polarity"],   
                           get\_predictions(estimator, predict\_train\_input\_fn))  
  with tf.Session() as session:  
    cm\_out = session.run(cm)  
  
# Normalize the confusion matrix so that each row sums to 1.  
cm\_out = cm\_out.astype(float) / cm\_out.sum(axis=1)[:, np.newaxis]  
  
sns.heatmap(cm\_out, annot=True, xticklabels=LABELS, yticklabels=LABELS);  
plt.xlabel("Predicted");  
plt.ylabel("True");

## 进一步改进

1. **情感回归**：我们使用分类器将每个例子分配到极性类中。但我们实际上还有另一个明确的特征 – 情感。这里的类实际上是一个比例，而基础值（正/负）可以很好地映射到一个连续的范围。我们可以利用这种属性计算回归（[DNN](https://tensorflow.google.cn/api_docs/python/tf/contrib/learn/DNNRegressor)回归[量](https://tensorflow.google.cn/api_docs/python/tf/contrib/learn/DNNRegressor)）来替代分类（[DNN分类](https://tensorflow.google.cn/api_docs/python/tf/contrib/learn/DNNClassifier)器）。
2. **更大的模块**：本教程中我们使用一个小模块来限制内存使用。有些具有更大的词汇量和更大的嵌入空间的模块可以提高准确度。
3. **参数调整**：我们可以通过调整元参数来提高准确度，例如学习速率或步数，特别是当我们使用不同的模块的时候。如果我们想获得合理的结果，验证集非常重要，因为设置一个学习预测训练数据的模型很容易，但推广到测试集却不很容易。
4. **更复杂的模型**：我们使用了一个通过计算每个句子中嵌入的单词，然后与它们的平均值相结合得到的模型。还可以使用线性模型（例如，[通用句子编码器](https://tfhub.dev/google/universal-sentence-encoder/2)模型）来更好地获取句子的性质。或者还可以是两个或多个TF-Hub函数的组合。
5. **正则化**：为了防止过度拟合，我们可以尝试使用执行某种正则化优化器，例如[Proximal Adagrad Optimizer](https://tensorflow.google.cn/api_docs/python/tf/train/ProximalAdagradOptimizer)。

## 高级：迁移学习分析

即使在**对小型数据集进行培训时，**迁移学习也可以**节省训练资源**并实现良好的模型归纳。在本部分中，我们将通过两个不同的TF-Hub模块进行训练来证明这一点：

* [**nnlm-en-dim128**](https://tfhub.dev/google/nnlm-en-dim128/1) - 预训练文本嵌入模块，
* [**随机nnlm烯dim128**](https://tfhub.dev/google/random-nnlm-en-dim128/1) -文本嵌入模块具有相同的词汇和网络**nnlm-dim128**，但权重只是随机初始化的数据，并没有受过真实数据训练。

通过以下两种模式进行培训：

* **仅**训练**分类器**（即不使用模型），和
* **使用模型**对**分类器进行**训练。

我们进行一些训练和评估，看看使用各种模型如何影响准确性。

def train\_and\_evaluate\_with\_module(hub\_module, train\_module=False):  
  embedded\_text\_feature\_column = hub.text\_embedding\_column(  
      key="sentence", module\_spec=hub\_module, trainable=train\_module)  
  
  estimator = tf.estimator.DNNClassifier(  
      hidden\_units=[500, 100],  
      feature\_columns=[embedded\_text\_feature\_column],  
      n\_classes=2,  
      optimizer=tf.train.AdagradOptimizer(learning\_rate=0.003))  
  
  estimator.train(input\_fn=train\_input\_fn, steps=1000)  
  
  train\_eval\_result = estimator.evaluate(input\_fn=predict\_train\_input\_fn)  
  test\_eval\_result = estimator.evaluate(input\_fn=predict\_test\_input\_fn)  
  
  training\_set\_accuracy = train\_eval\_result["accuracy"]  
  test\_set\_accuracy = test\_eval\_result["accuracy"]  
  
  return {  
      "Training accuracy": training\_set\_accuracy,  
      "Test accuracy": test\_set\_accuracy  
  }  
  
  
results = {}  
results["nnlm-en-dim128"] = train\_and\_evaluate\_with\_module(  
    "https://tfhub.dev/google/nnlm-en-dim128/1")  
results["nnlm-en-dim128-with-module-training"] = train\_and\_evaluate\_with\_module(  
    "https://tfhub.dev/google/nnlm-en-dim128/1", True)  
results["random-nnlm-en-dim128"] = train\_and\_evaluate\_with\_module(  
    "https://tfhub.dev/google/random-nnlm-en-dim128/1")  
results["random-nnlm-en-dim128-with-module-training"] = train\_and\_evaluate\_with\_module(  
    "https://tfhub.dev/google/random-nnlm-en-dim128/1", True)

我们来看看结果。

pd.DataFrame.from\_dict(results, orient="index")

|  | 测试准确性 | 训练准确性 |
| --- | --- | --- |
| nnlm烯dim128 | 0.79332 | 0.80304 |
| nnlm烯dim128-与模块培训 | 0.87108 | 0.94960 |
| 随机nnlm烯dim128 | 0.67704 | 0.72796 |
| 随机nnlm烯dim128-与模块培训 | 0.72104 | 0.76448 |

我们已经可以看到一些模式，但首先我们应该建立测试集的基线准确度 - 通过仅输出代表性最强的类的标签可以实现的精确度下限：

estimator.evaluate(input\_fn=predict\_test\_input\_fn)["accuracy\_baseline"]

0.5

在这里，我们指定代表性最高的类使我们的准确率达到**50％**。几点需要注意：

1. 也许令人惊讶的是，**模型仍然可以在固定的随机嵌入之上学习**。原因是即使字典中的每个单词都映射到随机向量，估计器也可以使用其完全连接层来分隔空间。
2. 与仅训练分类器不同的是允许用**随机嵌入**训练模型增加了训练集和测试集准确性。
3. 使用**预先训练的嵌入**对模型进行**训练**也可以提高精度。但请注意避免训练集上的过度拟合。即使在正则化的情况下训练预训练的模型也是危险的，因为嵌入权重无法代表在不同数据集上的模型，它们会变为新数据集上最理想的权重表示。

使用Estimator构建卷积神经网络

[tf.layers](https://tensorflow.google.cn/api_docs/python/tf/layers)模块提供了一个可以轻松构建神经网络高级API，还介绍了有助于创建密集（完全连接）层和卷积层、添加激活函数以及应用丢弃正则化的方法。在本教程中，您将学习如何使用layers构建卷积神经网络模型来识别MNIST数据集中的手写数字。

**所述**[**MNIST数据集**](http://yann.lecun.com/exdb/mnist/)**包括60000个训练样例，这些样本均为手写数字0-9，格式是28x28像素的单色图像。**

开始构建

让我们为TensorFlow程序设置框架。创建一个名为的文件 cnn\_mnist.py，并添加以下代码：

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
# Imports  
import numpy as np  
import tensorflow as tf  
  
tf.logging.set\_verbosity(tf.logging.INFO)  
  
# Our application logic will be added here  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  tf.app.run()

在学习本教程时，您将添加代码来构建，训练和评估卷积神经网络。完整的最终代码可以在 [这里找到](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/layers/cnn_mnist.py)。

卷积神经网络简介

卷积神经网络（CNN）是用于图像分类任务的当前最先进的模型架构。CNN将一系列滤器应用于图像的原始像素数据，以提取和学习更高级别的特征，然后该模型可用于分类。CNN包含三个组成部分：

* **卷积层**，将指定数量的卷积过滤器应用于图像。对于每个子区域，图层会执行一组数学运算，然后在输出特征图中生成单个值。然后，卷积层通常向输出应用 [ReLU激活函数](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))，以便将非线性规律引入模型中。
* **池化层**，[对](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer) 由卷积层提取[的图像数据](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer)进行 [下采样，](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer)以减少特征图的维数，从而减少处理时间。常用的池化算法是最大池化，他会提取特征图的子区域（例如，2×2像素块），保留子区域的最大值，并丢弃所有其他值。
* **密集（完全连接）层**，对卷积层提取并由池化层下采样的特征进行分类。在密集层中，层中的每个节点都连接到前一层中的每个节点。

通常，CNN由执行特征提取的多个卷积模块组成。每个模块由一个卷积层和一个池化层组成。最后一个卷积模块后是一个或多个执行分类的密集层。CNN中的最后一个密集层里，模型中每个目标类别的单个节点（模型可以预测的所有可能类）都都对应一个节点，并应用 [softmax](https://en.wikipedia.org/wiki/Softmax_function)激活函数为每个节点生成0-1之间的值（所有这些softmax值总和等于1）。我们可以将给定图像的softmax值解释为相对测量值，表明图像落入每个目标类的可能性。

**注意：**有关CNN架构的更全面的解说，请参阅斯坦福大学的[视觉识别卷积神经网络课程教材](https://cs231n.github.io/convolutional-networks/)。

构建CNN MNIST分类器

让我们使用以下CNN架构构建一个模型，对MNIST数据集中的图像进行分类：

1. **卷积层＃1**：应用32个5x5滤镜（提取5x5像素子区域），具有ReLU激活功能
2. **池化层＃1**：使用2x2过滤器和步长2执行最大池化运算（指定池化区域不重叠）
3. **卷积层＃2**：应用64个5x5过滤器，应用ReLU激活功能
4. **池化层＃2**：同样，使用2x2过滤器和步长2执行最大池化
5. **密集层＃1**：1,024个神经元，丢失正则化率为0.4（在训练期间任何给定元素将被丢弃的概率为0.4）
6. **密集层＃2（对数层）**：10个神经元，每个数字目标类一个（0-9）。

该[tf.layers](https://tensorflow.google.cn/api_docs/python/tf/layers)模块包含创建上述三种图层类型的方法：

* conv2d()。构造一个二维卷积层。接受的参数有过滤器数量，过滤器内核大小，填充和激活函数。
* max\_pooling2d()。构造一个使用最大池化算法的二维池化层。接受的参数有过滤器大小和步长大小。
* dense()。构造一个密集层。接受的参数有将神经元数量和激活函数。

这些方法中都接受张量作为输入并返回变换后的张量作为输出。这使得将一个层连接到另一个层变得容易：只需从一个层创建方法获取输出，并将其作为输入提供给另一个层即可。

打开cnn\_mnist.py并添加以下cnn\_model\_fn函数，该函数符合TensorFlow的Estimator API接口的要求（稍后将在[创建Estimator中详细介绍](https://tensorflow.google.cn/tutorials/estimators/cnn" \l "create_the_estimator)）。cnn\_mnist.py接受的参数为MNIST特征数据、标签和模型模式（来自 [tf.estimator.ModeKeys](https://tensorflow.google.cn/api_docs/python/tf/estimator/ModeKeys)：TRAIN，EVAL，PREDICT）; 配置CNN，并返回预测、损失和训练操作：

def cnn\_model\_fn(features, labels, mode):  
  """Model function for CNN."""  
  # Input Layer  
  input\_layer = tf.reshape(features["x"], [-1, 28, 28, 1])  
  
  # Convolutional Layer #1  
  conv1 = tf.layers.conv2d(  
      inputs=input\_layer,  
      filters=32,  
      kernel\_size=[5, 5],  
      padding="same",  
      activation=tf.nn.relu)  
  
  # Pooling Layer #1  
  pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)  
  
  # Convolutional Layer #2 and Pooling Layer #2  
  conv2 = tf.layers.conv2d(  
      inputs=pool1,  
      filters=64,  
      kernel\_size=[5, 5],  
      padding="same",  
      activation=tf.nn.relu)  
  pool2 = tf.layers.max\_pooling2d(inputs=conv2, pool\_size=[2, 2], strides=2)  
  
  # Dense Layer  
  pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])  
  dense = tf.layers.dense(inputs=pool2\_flat, units=1024, activation=tf.nn.relu)  
  dropout = tf.layers.dropout(  
      inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)  
  
  # Logits Layer  
  logits = tf.layers.dense(inputs=dropout, units=10)  
  
  predictions = {  
      # Generate predictions (for PREDICT and EVAL mode)  
      "classes": tf.argmax(input=logits, axis=1),  
      # Add `softmax\_tensor` to the graph. It is used for PREDICT and by the  
      # `logging\_hook`.  
      "probabilities": tf.nn.softmax(logits, name="softmax\_tensor")  
  }  
  
  if mode == tf.estimator.ModeKeys.PREDICT:  
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)  
  
  # Calculate Loss (for both TRAIN and EVAL modes)  
  loss = tf.losses.sparse\_softmax\_cross\_entropy(labels=labels, logits=logits)  
  
  # Configure the Training Op (for TRAIN mode)  
  if mode == tf.estimator.ModeKeys.TRAIN:  
    optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.001)  
    train\_op = optimizer.minimize(  
        loss=loss,  
        global\_step=tf.train.get\_global\_step())  
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train\_op=train\_op)  
  
  # Add evaluation metrics (for EVAL mode)  
  eval\_metric\_ops = {  
      "accuracy": tf.metrics.accuracy(  
          labels=labels, predictions=predictions["classes"])}  
  return tf.estimator.EstimatorSpec(  
      mode=mode, loss=loss, eval\_metric\_ops=eval\_metric\_ops)

以下部分（标题对应于上面的每个代码块）详细介绍了用于创建每个图层的[tf.layers](https://tensorflow.google.cn/api_docs/python/tf/layers)代码，以及如何计算损失、配置训练操作和生成预测。如果您已经在熟悉CNN[TensorFlow Estimator](https://tensorflow.google.cn/guide/custom_estimators)，并认为上面的代码直观易懂，那可以跳过这些部分或直接跳到[“培训和评估CNN MNIST分类”](https://tensorflow.google.cn/tutorials/estimators/cnn#train_eval_mnist)。

输入层

默认情况下，layers模块中用于为二维图像数据创建卷积和池化层的的方法输入张量形状应该为[*batch\_size*, *image\_height*, *image\_width*,*channels*]。可以使用*data\_format*参数更改此行为; 定义如下：

* *batch\_size*。在训练期间执行梯度下降时使用的样本子集的大小。
* *image\_height*。示例图像的高度。
* *image\_width*。示例图像的宽度。
* *channels*。示例图像中的颜色通道数。对于彩色图像，通道数为3（红色，绿色，蓝色）。对于单色图像，只有1个通道（黑色）。
* *data\_format*。一个字符串， channels\_last（默认）或channels\_first。channels\_last对应于形状为 (batch, ..., channels)的输入，而channels\_first对应于具有形状为(batch, channels, ...)的输入。

在这里，我们的MNIST数据集由单色28x28像素图像组成，因此我们输入层的所需形状是。[*batch\_size*, 28, 28, 1]

要将输入特征图（features）转换为此形状，我们可以执行以下reshape操作：

input\_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

请注意，我们已指出批量大小为-1，它表示应根据features["x"]输入值的数量动态计算此维度，并保持所有其他维度的大小不变。这使得我们可以将batch\_size视为可以调整的超参数。例如，如果我们按照批次大小5样本馈送到模型中，则features["x"]将包含3,920个值（每个图像中每个像素一个值），并且input\_layer其形状为[5, 28, 28, 1]。同样，如果我们批次大小为100，features["x"]将包含78,400个值，并且input\_layer其形状为[100, 28, 28, 1]。

卷积层＃1

在我们的第一个卷积层中，我们希望将32个5x5过滤器应用于输入层，并使用ReLU激活功能。我们可以使用 layers模块中的conv2d()方法创建此层，如下所示：

conv1 = tf.layers.conv2d(  
    inputs=input\_layer,  
    filters=32,  
    kernel\_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)

该inputs参数指定了输入张量，其中张量必须有[*batch\_size*, *image\_height*, *image\_width*,*channels*]的形状 。在这里，我们将第一个卷积层连接到具有形状为[*batch\_size*, 28, 28, 1]的。

**注意：**在传递参数**data\_format=channels\_first**时，**conv2d()接受**的形状为  **[*batch\_size*, *channels*,*image\_height*, *image\_width*]。**

filters参数指定的过滤器数量（本教程中为32）， kernel\_size指定了过滤器的维度（本教程中为[5, 5]）。

**提示：**如果过滤器高度和宽度具有相同的值，则可以为kernel\_size 指定单个整数（例如kernel\_size=5）。

padding参数指定了两个枚举值（不区分大小写）中的一个：valid（默认值）或same。要指定输出张量应具有与输入张量相同的高度和宽度值，我们在此处设置padding=same，它指示TensorFlow将0值添加到输入张量的边缘以保持28的高度和宽度。（没有填充的话，在28x28张量上进行5x5卷积将产生24x24张量，因为在28x28网格中有24x24个位置可以提取5x5图块。）

activation参数指定了应用于卷积输出的激活函数。在这里，我们使用[tf.nn.relu](https://tensorflow.google.cn/api_docs/python/tf/nn/relu)指定作为ReLU激活函数。

conv2d()生成的输出张量具有以下形状 ：，与输入具有相同的高度和宽度尺寸，但现在有32个通道，用于保存每个过滤器的输出。

池化层＃1

接下来，我们将第一个池化层连接到刚刚创建的卷积层。我们可以使用layers中的max\_pooling2d()方法构造一个层，该层使用2x2过滤器和步长为2执行最大池化运算：

pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)

同样，inputs指定输入张量，形状为 [*batch\_size*, *image\_height*, *image\_width*, *channels*]。这里，我们的输入张量是第一个卷积层的输出conv1，其形状为 [*batch\_size*, 28, 28, 32]。

**注意：**和**conv2d()一样**，**max\_pooling2d()**将接受传递参数**data\_format=channels\_first**时将接受**[*batch\_size*,*channels*, *image\_height*, *image\_width*]这种**形状**。**

pool\_size参数指定了最大池过滤器的大小为 [*height*, *width*]（这里为[2, 2]）。如果两个维度具有相同的值，则可以改为指定单个整数（例如，pool\_size=2 ）。

strides参数指定步幅的大小。在这里，我们设置了步幅为2，表示过滤器提取的子区域在高度和宽度维度上应分开2个像素（对于2x2过滤器，这意味着所提取的区域都不会重叠）。如果要为高度和宽度设置不同的步幅值，可以改为指定元组或列表（例如，stride=[3, 6]）。

max\_pooling2d()（pool1）生成的输出张量具有以下形状 ：2x2滤波器将高度和宽度各减少50％。

卷积层＃2和池层＃2

我们可以使用conv2d()和max\_pooling2d()将第二个卷积和池化层连接到CNN 。对于卷积层＃2，我们使用ReLU激活配置64个5x5过滤器，对于池＃2，我们使用与池化层1相同的规格（2x2最大池过滤器，步幅为2）：

conv2 = tf.layers.conv2d(  
    inputs=pool1,  
    filters=64,  
    kernel\_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)  
  
pool2 = tf.layers.max\_pooling2d(inputs=conv2, pool\_size=[2, 2], strides=2)

注意，卷积层＃2将我们的第一个池化层（pool1）的输出张量作为输入，并将张量conv2作为输出。conv2 具有与pool1相同的高度和宽度的形状（因为padding="same"），以及用于64个滤波器的64个通道，形状为[*batch\_size*, 14, 14, 64] 。

池化层＃2 conv2作为输入，产生pool2输出。pool2 有形状（conv2高度和宽度减少50％）

密集层

接下来，我们想要向我们的CNN添加密集层（具有1,024个神经元和ReLU激活函数），以对卷积/池化层提取的特征进行分类。不过在我们连接图层之前，我们会先扁平化特征图 (pool2)，以将其变形为[*batch\_size*, *features*]，使得我们的张量只有两个维度：

pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])

在reshape()上面的操作中，-1表示*batch\_size* 将根据输入数据中的示例数量动态计算维度。每个示例都有7（pool2高度）\* 7（pool2宽度）\* 64（pool2通道）个特征，因此我们希望features维度的值为7 \* 7 \* 64（总共3136）。输出张量pool2\_flat具有的形状 。

现在，我们可以使用layers的dense()方法连接到密集层，如下所示：

dense = tf.layers.dense(inputs=pool2\_flat, units=1024, activation=tf.nn.relu)

该inputs参数指定输入张量：扁平化特征图 pool2\_flat。该units参数指定密集层中神经元的数目（1024）。activation参数表示采用的激活函数; 同样，我们将使用[tf.nn.relu](https://tensorflow.google.cn/api_docs/python/tf/nn/relu)添加ReLU激活。

为了改善模型的结果，我们还使用layers中的dropout方法，向密集层应用丢弃正则化：

dropout = tf.layers.dropout(  
    inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

inputs指定输入张量，它是来自密集层（dense）的输出张量。

rate参数指定了丢弃率; 在这里，我们使用0.4，这意味着40％的元素将在训练期间被随机丢弃。

training参数采用布尔值表示模型目前是否在训练模式下运行; 如果training为 True，则仅执行丢失操作。在这里，我们检查传递给我们的模型函数 cnn\_model\_fn的mode是否是TRAIN模式。

输出张量dropout形状应该为[*batch\_size*, 1024]。

对数层

我们的神经网络中的最后一层是对数层，它将返回我们预测的原始值。我们创建了一个包含10个神经元的密集层（介于 0 到 9 之间的每个目标类别对应一个神经元），并应用线性激活函数（默认函数）：

logits = tf.layers.dense(inputs=dropout, units=10)

CNN的最终输出张量logits形状为 。

生成预测

模型的对数层以[*batch\_size*, 10] 维张量中原始值的形式返回预测。。让我们将这些原始值转换为我们的模型函数可以返回的两种不同格式：

* 每个例子的**预测类**：0-9的数字。
* 每个样本属于每个可能的目标类的概率：该样本是0，是1，是2等的概率

对于给定的示例，我们的预测类是具有最高原始值的相应行中的元素。我们可以使用[tf.argmax](https://tensorflow.google.cn/api_docs/python/tf/argmax) 函数找到这个元素的索引：

tf.argmax(input=logits, axis=1)

input参数指定从中提取最大值，这里的张量logits。该axis参数指定input 张量沿找到最大值的轴。在这里，我们需要沿着在索引为1的维度上找到最大值，这对应于我们的预测（已知我们的张量形状为）。

我们可以通过使用[tf.nn.softmax](https://tensorflow.google.cn/api_docs/python/tf/nn/softmax)的softmax激活函数来从对数层得出概率：

tf.nn.softmax(logits, name="softmax\_tensor")

**注意：**我们使用**name**参数来明确命名此操作为 **softmax\_tensor**，便于我们稍后可以引用它。（我们将在[“设置日志记录钩子”部分中设置](https://tensorflow.google.cn/tutorials/estimators/cnn" \l "set_up_a_logging_hook) softmax值[的日志记录](https://tensorflow.google.cn/tutorials/estimators/cnn" \l "set_up_a_logging_hook)）。

我们将预测编译为字典，并返回一个EstimatorSpec对象：

predictions = {  
    "classes": tf.argmax(input=logits, axis=1),  
    "probabilities": tf.nn.softmax(logits, name="softmax\_tensor")  
}  
if mode == tf.estimator.ModeKeys.PREDICT:  
  return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

计算损失

对于训练和评估，我们需要定义一个 [损失函数](https://en.wikipedia.org/wiki/Loss_function) 来衡量模型预测与目标类别的匹配程度。对于像MNIST这样的多类分类问题，通常用 [交叉熵](https://en.wikipedia.org/wiki/Cross_entropy)作为损失指标。以下代码计算模型以TRAIN或EVAL模式运行时交叉熵：

loss = tf.losses.sparse\_softmax\_cross\_entropy(labels=labels, logits=logits)

让我们详细了解一下上述代码。

labels张量包含样本的预测索引列表，例如[1, 9, ...]。logits包含最后一层的线性输出。

[tf.losses.sparse\_softmax\_cross\_entropy](https://tensorflow.google.cn/api_docs/python/tf/losses/sparse_softmax_cross_entropy)，以高效的数值稳定方式从这两个输入计算softmax交叉熵（又名：分类交叉熵，负对数似然率）。

配置培训操作

在上一节中，我们将CNN的损失定义为对数层和标签的softmax交叉熵。下面让我们配置模型以在训练期间优化这个损失值。我们将使用0.001的学习率和 [随机梯度下降](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) 的优化算法：

if mode == tf.estimator.ModeKeys.TRAIN:  
  optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.001)  
  train\_op = optimizer.minimize(  
      loss=loss,  
      global\_step=tf.train.get\_global\_step())  
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train\_op=train\_op)

**注意：**有关为Estimator模型函数配置训练操作的更深入了解，请参阅[“](https://tensorflow.google.cn/guide/custom_estimators#defining-the-training-op-for-the-model)[在tf.estimator中创建估计”](https://tensorflow.google.cn/guide/custom_estimators)教程 中的[“](https://tensorflow.google.cn/guide/custom_estimators)[为模型定义训练操作](https://tensorflow.google.cn/guide/custom_estimators#defining-the-training-op-for-the-model)[”](https://tensorflow.google.cn/guide/custom_estimators)。

添加评估指标

要在我们的模型中添加准确度指标，我们在EVAL模式中定义eval\_metric\_ops 字典，如下所示：

eval\_metric\_ops = {  
    "accuracy": tf.metrics.accuracy(  
        labels=labels, predictions=predictions["classes"])}  
return tf.estimator.EstimatorSpec(  
    mode=mode, loss=loss, eval\_metric\_ops=eval\_metric\_ops)

训练和评估CNN MNIST分类器

我们编写了MNIST CNN模型函数; 现在我们准备进行训练和评估。

加载训练和测试数据

首先，加载我们的培训和测试数据。使用以下代码将main()函数添加到 cnn\_mnist.py：

def main(unused\_argv):  
  # Load training and eval data  
  mnist = tf.contrib.learn.datasets.load\_dataset("mnist")  
  train\_data = mnist.train.images # Returns np.array  
  train\_labels = np.asarray(mnist.train.labels, dtype=np.int32)  
  eval\_data = mnist.test.images # Returns np.array  
  eval\_labels = np.asarray(mnist.test.labels, dtype=np.int32)

我们在 train\_data 和 train\_labels 中将训练特征数据（55000个图像的原始像素值）和训练标签（每张图像在 0 到 9 中的对应值）分别储存为[numpy的阵列](https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html) 。同样，我们将评估特征数据（10,000个图像）和评估标签分别存储在eval\_data 和中eval\_labels。

创建Estimator

接下来，让我们为我们的模型创建一个Estimator（用于执行高级模型训练，评估和推理的TensorFlow类）。将以下代码添加到main()：

# Create the Estimator  
mnist\_classifier = tf.estimator.Estimator(  
    model\_fn=cnn\_model\_fn, model\_dir="/tmp/mnist\_convnet\_model")

model\_fn参数指定用于训练、评估和预测的模型函数; 我们在[“构建CNN MNIST分类器”中](https://tensorflow.google.cn/tutorials/estimators/cnn#building_the_cnn_mnist_classifier)创建了的cnn\_model\_fn 传递到该参数[。](https://tensorflow.google.cn/tutorials/estimators/cnn#building_the_cnn_mnist_classifier) model\_dir参数指定了用于保存模型数据（检查点）的目录（这里，我们指定的临时目录为/tmp/mnist\_convnet\_model，但您可以随时更改选择的另一个目录）。

**注意：**有关TensorFlow **Estimator**API 的消息内容，请参阅教程[“在tf.estimator中创建估算器”。](https://tensorflow.google.cn/guide/custom_estimators)

设置日志记录钩子

由于CNN可能需要一段时间才能进行训练，我们可以设置一些记录，以便我们可以在训练期间跟踪进度。我们可以使用TensorFlow 的[tf.train.SessionRunHook](https://tensorflow.google.cn/api_docs/python/tf/train/SessionRunHook)创建一个[tf.train.LoggingTensorHook](https://tensorflow.google.cn/api_docs/python/tf/train/LoggingTensorHook)， 记录来自CNN的softmax层的概率值。将以下内容添加到main()：

# Set up logging for predictions  
tensors\_to\_log = {"probabilities": "softmax\_tensor"}  
logging\_hook = tf.train.LoggingTensorHook(  
    tensors=tensors\_to\_log, every\_n\_iter=50)

我们存储了将要记录的张量的词典tensors\_to\_log。每个键都是我们选择的将会显示在日志输出中的标签，相应的标签是TensorFlow图中的Tensor的名称。在这里，我们可以在softmax\_tensor找到probabilities（我们在cnn\_model\_fn生成概率时提供softmax操作的名称）。

**注意：**如果未通过**name** 参数为操作明确指定名称，TensorFlow将指定默认名称。可以通过下面两种简单的方法找到应用于操作的名称：在[TensorBoard](https://tensorflow.google.cn/guide/graph_viz)上可视化您的图形 或启用 [TensorFlow调试器（tfdbg）](https://tensorflow.google.cn/guide/debugger)。

接下来，我们创建LoggingTensorHook，传递tensors\_to\_log传递给 tensors参数。我们设置every\_n\_iter=50，它指定在每50个训练步骤之后记录概率。

训练模型

现在我们可以训练模型了，可以通过创建train\_input\_fn 和调用train()来完成mnist\_classifier。将以下代码添加到main()：

# Train the model  
train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    x={"x": train\_data},  
    y=train\_labels,  
    batch\_size=100,  
    num\_epochs=None,  
    shuffle=True)  
mnist\_classifier.train(  
    input\_fn=train\_input\_fn,  
    steps=20000,  
    hooks=[logging\_hook])

在numpy\_input\_fn调用中，我们将训练特征数据和标签分别传递给 x（作为字典）和y。我们设置batch\_size为100（这意味着该模型会在每一步训练 100 个小批次样本）。 num\_epochs=None表示模型将训练直到达到指定的步数。我们还设置shuffle=True，以随机化处理训练数据。在train调用中，我们设置steps=20000 （这意味着模型将训练总共20,000步）。为了在训练期间触发 logging\_hook，我们将其传递到 hooks 参数。

评估模型

培训完成后，我们要评估我们的模型，以确定其在MNIST测试集上的准确性。我们调用该evaluate方法，该方法评估我们在model\_fn的eval\_metric\_ops参数中指定的指标。将以下内容添加到main()：

# Evaluate the model and print results  
eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    x={"x": eval\_data},  
    y=eval\_labels,  
    num\_epochs=1,  
    shuffle=False)  
eval\_results = mnist\_classifier.evaluate(input\_fn=eval\_input\_fn)  
print(eval\_results)

为了创建eval\_input\_fn，我们进行设置num\_epochs=1，以便评估模型在一个数据周期上的指标并返回结果。我们还设置 shuffle=False以按顺序迭代数据。

运行模型

我们编写了CNN模型函数Estimator，以及训练/评估逻辑; 现在让我们看看结果。运行cnn\_mnist.py。

**注意：**训练CNN的计算量非常大。**cnn\_mnist.py的**预计完成时间取决于您的处理器，但在CPU上可能会超过1小时。要更快地训练，您可以减少**steps**传递的次数**train()**，但请注意，这会影响准确性。

当模型训练时，您将看到如下的日志输出：

INFO:tensorflow:loss = 2.36026, step = 1  
INFO:tensorflow:probabilities = [[ 0.07722801  0.08618255  0.09256398, ...]]  
...  
INFO:tensorflow:loss = 2.13119, step = 101  
INFO:tensorflow:global\_step/sec: 5.44132  
...  
INFO:tensorflow:Loss for final step: 0.553216.  
  
INFO:tensorflow:Restored model from /tmp/mnist\_convnet\_model  
INFO:tensorflow:Eval steps [0,inf) for training step 20000.  
INFO:tensorflow:Input iterator is exhausted.  
INFO:tensorflow:Saving evaluation summary for step 20000: accuracy = 0.9733, loss = 0.0902271  
{'loss': 0.090227105, 'global\_step': 20000, 'accuracy': 0.97329998}

在这里，我们的测试数据集的准确率达到了97.3％。

其他资源

要了解有关TensorFlow中TensorFlow Estimators和CNN的更多信息，请参阅以下资源：

* [在tf.estimator中创建Estimators](https://tensorflow.google.cn/guide/custom_estimators) 提供了对TensorFlow Estimator API的介绍。它说明了如何配置Estimator，编写模型函数，计算损失以及定义训练操作。
* [高级卷积神经网络](https://tensorflow.google.cn/tutorials/images/deep_cnn)通过使用较低级别的TensorFlow操作，演示如何构建*无需估计器的*MNIST CNN分类模型。