

Estimator 方法中进行模型的构建。例如：

```
config = tf.estimator.RunConfig(train_distribute=distribution)
classifier = tf.estimator.Estimator(model_fn=model_fn, config=config)
```

在使用多机训练的分布策略时，还需要指定网络中的角色关系。更多例子可参考以下链接：

<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/distribute/README.md>

## 6.1.11 用 tfdbg 调试 TensorFlow 模型

在 TensorFlow 中提供了可以调试程序的 API——tfdbg。用 tfdbg 可以轻松地对原生的 TensorFlow 程序、TF-slim 程序、Estimators 程序、tf.keras 程序、TFLearn 程序进行调试。官网上提供了详细的文档教程。具体链接如下：

[https://www.tensorflow.org/programmers\\_guide/debugger](https://www.tensorflow.org/programmers_guide/debugger)

在该链接中，介绍了用 tfdbg 调试一个训练过程中生成 inf 和 nan 值的例子。这也是 tfdbg 的重要价值所在。由于篇幅原因，这里不再详细介绍。读者可以跟着该网站教程自行学习。

TensorFlow 中还提供了配合 tfdbg 的可视化插件，该插件可以集成到 Tensorboard 中进行使用。具体说明见以下链接：

<https://github.com/tensorflow/tensorboard/blob/master/tensorboard/plugins/debugger/README.md>

## 6.1.12 用钩子函数（Training\_Hooks）跟踪训练状态

在 TensorFlow 中有一个 Training\_Hooks 接口，它实现了钩子函数的功能。该接口由多种 API 组成。在程序中使用 Training\_Hooks 接口，可以跟踪模型在训练、运行过程中各个环节的具体的状态。该接口的说明见表 6-1。

表 6-1 Training\_Hooks 接口的说明

接口名称	描述
tf.train.SessionRunHook	所有钩子函数的基类。若想自定义钩子函数，则可以集成该类。更多信息参考： <a href="https://www.tensorflow.org/api_docs/python/tf/train/SessionRunHook">https://www.tensorflow.org/api_docs/python/tf/train/SessionRunHook</a>
tf.train.LoggingTensorHook	按照指定步数输出指定张量的值。这是十分常用的钩子函数。更多信息参考： <a href="https://www.tensorflow.org/api_docs/python/tf/train/LoggingTensorHook">https://www.tensorflow.org/api_docs/python/tf/train/LoggingTensorHook</a>
tf.train.StopAtStepHook	在指定步数之后停止跟踪。更多信息参考： <a href="https://www.tensorflow.org/api_docs/python/tf/train/StopAtStepHook">https://www.tensorflow.org/api_docs/python/tf/train/StopAtStepHook</a>
tf.train.CheckpointSaverHook	按照指定步数或时间生成检查点文件。还可以用 tf.train.CheckpointSaverListener 函数监听生成检查点文件的操作，并可以在操作过程的前、中、后 3 个阶段设置回调函数。更多信息参考： <a href="https://www.tensorflow.org/api_docs/python/tf/train/CheckpointSaverHook">https://www.tensorflow.org/api_docs/python/tf/train/CheckpointSaverHook</a>

续表

Training_Hooks 的接口名称	描述
tf.train.StepCounterHook	按照指定步数或时间计数。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/StepCounterHook">https://www.tensorflow.org/api_docs/python/tf/train/StepCounterHook</a>
tf.train.NanTensorHook	指定要监视的 loss 张量。如果 loss 为 NaN，则停止运行。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/NanTensorHook">https://www.tensorflow.org/api_docs/python/tf/train/NanTensorHook</a>
tf.train.SummarySaverHook	按照指定步数保存摘要信息。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/SummarySaverHook">https://www.tensorflow.org/api_docs/python/tf/train/SummarySaverHook</a>
tf.train.GlobalStepWaiterHook	直到 Global step 的值达到指定值后才开始执行。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/GlobalStepWaiterHook">https://www.tensorflow.org/api_docs/python/tf/train/GlobalStepWaiterHook</a>
tf.train.FinalOpsHook	获取某个张量在会话（session）结束时的值。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/FinalOpsHook">https://www.tensorflow.org/api_docs/python/tf/train/FinalOpsHook</a>
tf.train.FeedFnHook	指定输入，并获取输入信息的钩子函数。更多信息参考: <a href="https://www.tensorflow.org/api_docs/python/tf/train/FeedFnHook">https://www.tensorflow.org/api_docs/python/tf/train/FeedFnHook</a>
tf.train.ProfilerHook	捕获硬件运行时的分配信息。更多信息参考: <a href="https://github.com/catapult-project/catapult/blob/master/tracing/README.md">https://github.com/catapult-project/catapult/blob/master/tracing/README.md</a>

表 6-1 中的钩子（Hook）类一般会配合 tf.train.MonitoredSession 一起使用，有时也会配合估算器一起使用。在本书 6.4.12 小节会通过详细实例来演示其用法。

想了解更多信息，还可以参考官方文档：

[https://www.tensorflow.org/api\\_guides/python/train#Training\\_Hooks](https://www.tensorflow.org/api_guides/python/train#Training_Hooks)

### 6.1.13 用分布式运行方式训练模型

在大型的数据集上训练神经网络，需要的运算资源非常大，而且还要花上很长时间才能完成。

为了缩短训练时间，可以用分布式部署的方式将一个训练任务拆成多个小任务，分配到不同的计算机上，来完成协同运算。这样用计算机群运算来代替单机计算，可以使训练时间大大变短。

TensorFlow 1.4 版本之后的估算器具有 train\_and\_evaluate 函数。该函数可以使分布式训练的实现变得更为简单。只需要修改 TF\_CONFIG 环境变量（或在程序中指定 TF\_CONFIG 变量），即可实现分布式中不同的角色的协同合作，具体可见 6.9 节。

### 6.1.14 用 T2T 框架系统更方便地训练模型

Tensor2Tensor（T2T）是谷歌开源的一个模块化深度学习框架，其中包含当前各个领域中最先进的模型，以及训练模型时常用到的数据集。

#### 1. T2T 框架的详细介绍

T2T 框架构建在 TensorFlow 之上。在 T2T 框架中定义了深度学习系统所需的各个组件：数据集、模型架构、优化器、学习速率衰减方案、超参数等。

每个组件中都采用了目前最好的机器学习方法，例如：序列填充（padding）、计算交叉熵损失、用调试好的 Adam 优化器参数、自适应批处理、同步的分布式训练、调试好的图像数据增强、标签平滑和大量的超参数配置等。

组件彼此之间统一采用标准化接口，形成模块化的架构。使用者只需选择数据集、模型、优化器并设定好超参数，就可以实现训练模型、查看性能等操作。

另外，在整个模块化架构中，每个组件都是通过一个函数来实现的。每个函数的输入和输出都是一个标准格式的张量，以便使用者用自定义组件对现有组件进行替换。

## 2. T2T 框架的使用环境

T2T 框架主要用于谷歌的 TPU 开发环境，当然也可用于本地开发环境。

用 T2T 直接在云端进行训练，可以使研究者不再需要花费昂贵的成本购买硬件，为用户带来更便捷的体验。但是这种方式的弊端是——过分依赖网络。

本书只介绍 T2T 框架在本地环境下的使用。有关云端的使用方式，需要读者自行研究。

## 3. T2T 的环境搭建

T2T 的代码独立于 TensorFlow 主框架，需要单独安装，具体命令如下：

```
pip install tensor2tensor
```

如想了解更多关于 T2T 的细节，可以在以下链接中查看 T2T 框架的源码及教程：

```
https://github.com/tensorflow/tensor2tensor
```

有关 T2T 框架的使用实例，见本书 6.11 节、6.12 节。

### 6.1.15 将TensorFlow 1.x 中的代码移植到2.x 版本

在 TensorFlow 2.x 版本中，提供了一个升级 TensorFlow 1.x 版本代码的工具——`tf_upgrade_v2`。该工具可以非常方便地将 TensorFlow 1.x 版本中编写的代码移植到 TensorFlow 2.x 中。

`tf_upgrade_v2` 工具支持单文件转换和多文件批量转换两种方式。

#### 1. 对单个代码文件进行转换

在命令行里输入 `tf_upgrade_v2` 命令，用“`--infile`”参数来指定输入文件，用“`--outfile`”参数来指定输出文件。具体命令如下：

```
tf_upgrade_v2 --infile foo_v1.py --outfile foo_v2.py
```

该命令可以将 TensorFlow 1.x 版本中编写的代码文件 `foo_v1.py` 转成可以支持 TensorFlow 2.x 版本的代码 `foo_v2.py`。

#### 2. 批量转化多个代码文件

在命令行里输入 `tf_upgrade_v2` 命令，用“`-intree`”参数来指定输入文件路径，用“`-outtree`”参数来指定输出文件路径。具体命令如下：

```
tf_upgrade_v2 -intree foo_v1 -outtree foo_v2
```

该命令可以将目录为 `foo_v1` 下的所有代码文件转成支持 TensorFlow 2.x 版本的代码文件，并保存到目录 `foo_v2` 中。



### 提示：

虽然 `tf_upgrade_v2` 工具的转化功能能解决大部分的移植工作，但是对于一些特殊的 API 仍需要手动来移植。例如：

TensorFlow 2.x 版本中不再有 TensorFlow 1.x 版本中的 `tf.contrib` 模块。

在 TensorFlow 2.x 版本中，TensorFlow 1.x 版本中的 `tf.contrib` 模块被拆分成两部分：

- 一部分被移植到 TensorFlow 2.x 版本的主框架下，可以用 `tf_upgrade_v2` 工具进行转化。
- 一部分将被移除，无法被转化。在升级代码时，需要手动编写代码。

另外，在 TensorFlow 1.x 版本中带有废弃标注的 API，也不会出现在 TensorFlow 2.x 版本中。这些转化失败的 API 都需要被替换成推荐使用的 API。

具体转化实例见本书 6.13 节。

## 6.1.16 TensorFlow 2.x 中的新特性——自动图

在 TensorFlow 1.x 版本中，要开发基于张量控制流的程序，必须使用 `tf.conf`、`tf.while_loop` 之类的专用函数。这增加了开发的复杂度。

在 TensorFlow 2.x 版本中，可以通过自动图（AutoGraph）功能，将普通的 Python 控制流语句转成基于张量的运算图。这大大简化了开发工作。

在 TensorFlow 2.x 版本中，可以用 `tf.function` 装饰器修饰 Python 函数，将其自动转化成张量运算图。示例代码如下：

```
import tensorflow as tf #导入TensorFlow2.0
@tf.function #用自动图修饰的函数
def autograph(input_data):
    if tf.reduce_mean(input_data) > 0:
        return input_data #返回是整数类型
    else:
        return input_data // 2 #返回整数类型
a =autograph(tf.constant([-6, 4])) #在TensorFlow 2.x 上运行，输出：[-3 2] [6-4]
b =autograph(tf.constant([6, -4]))
print(a.numpy(),b.numpy())
```

从上面代码的输出结果中可以看到，程序运行了控制流 “`tf.reduce_mean(input_data) > 0`” 语句的两个分支。这表明被装饰器 `tf.function` 修饰的函数具有张量图的控制流功能。



### 提示：

在使用自动图功能时，如果在被修饰的函数中有多个返回分支，则必须确保所有的分支都返回相同类型的张量，否则会报错。

## 6.2 实例 14：用静态图训练一个具有保存检查点功能的回归模型

本节用一个简单的模型来演示静态图的使用方法。

### 实例描述

假设有一组数据集，其中  $x$  和  $y$  的对应关系  $y \approx 2x$ 。

本实例就是让神经网络学习这些样本，并找到其中的规律，即让神经网络自己能够总结出  $y \approx 2x$  这样的公式。

在训练的过程中将生成检查点文件，并在程序结束之后二次载入检查点文件，接着训练。

本实例属于一个回归任务。回归任务是指，对输入数据进行计算，并输出某个具体值的任务。与之相对的还有分类任务，它们都是深度学习中最常见的任务模式。这部分内容在第 7 章特征工程中还会重点介绍。

### 6.2.1 准备开发步骤

在实现过程中，需要完成的具体步骤如下：（1）生成模拟样本；（2）搭建全连接网络模型；（3）训练模型。其中，在第（3）步训练模型过程中，还需要完成对检查点文件的生成和载入。



#### 提示：

全连接网络是最基础的神经网络模型。它是将上层的网络节点与下层的网络节点全部连接起来。该结构可以通过增加网络节点个数的方式，实现拟合任意数据分布的效果，但是过多的节点又会降低模型的计算性能与泛化性。

有关全接网络的更多使用可以参考 7.2 节的 wide\_deep 模型。

有关全接网络的更多原理可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的第 6、7 章。

### 6.2.2 生成检查点文件

在生成检查点文件时，步骤如下：

（1）实例化一个 saver 对象。

（2）在会话（session）中，调用 saver 对象的 save 方法保存检查点文件。

#### 1. 生成 saver 对象

saver 对象是由 tf.train.Saver 类的实例化方法生成的。该方法有很多参数，常用的有以下几个。

- `var_list`: 指定要保存的变量。
- `max_to_keep`: 指定要保留检查点文件的个数。
- `keep_checkpoint_every_n_hours`: 指定间隔几小时保存一次模型。

实例代码如下：

```
saver = tf.train.Saver(tf.global_variables(), max_to_keep=1)
```

该代码表示将全部的变量保存起来。最多只保存一个检查点文件（一个检查点文件包含 3 个子文件）。

## 2. 生成检查点文件

调用 `saver` 对象的 `save` 生成保存检查点文件。实例代码如下：

```
saver.save(sess, savedir+"linermodel.cpkt", global_step=epoch)
```

该代码运行后，系统会将检查点文件保存到 `savedir` 路径。同时，也将迭代次数 `global_step` 的值放到了检查点文件的名字中。

### 6.2.3 载入检查点文件

首先用 `tf.train.latest_checkpoint` 方法找到最近的检查点文件，接着用 `saver.restore` 方法将该检查点文件载入。实例代码如下：

```
kpt = tf.train.latest_checkpoint(savedir)      #找到最近的检查点文件
if kpt!=None:
    saver.restore(sess, kpt)                  #载入检查点文件
```

### 6.2.4 代码实现：在线性回归模型中加入保存检查点功能

在代码第 37 行，定义了一个 `saver` 张量。在会话运行中，用 `saver` 对象的 `save` 方法来生成检查点文件（见代码第 66、69 行）。

具体代码如下：

#### 代码 6-1 用静态图训练一个具有保存检查点功能的回归模型

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04 print(tf.__version__)
05 # (1) 生成模拟数据
06 train_X = np.linspace(-1, 1, 100)
07 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 #y=2x, 但是加入了噪声
08 #图形显示
09 plt.plot(train_X, train_Y, 'ro', label='Original data')
10 plt.legend()
11 plt.show()
12
```

```

13 tf.reset_default_graph()
14
15 # (2) 构建模型
16
17 #构建模型
18 #占位符
19 X = tf.placeholder("float")
20 Y = tf.placeholder("float")
21 #模型参数
22 W = tf.Variable(tf.random_normal([1]), name="weight")
23 b = tf.Variable(tf.zeros([1]), name="bias")
24 #前向结构
25 z = tf.multiply(X, W)+ b
26 global_step = tf.Variable(0, name='global_step', trainable=False)
27 #反向优化
28 cost =tf.reduce_mean(tf.square(Y - z))
29 learning_rate = 0.01
30 optimizer =
31     tf.train.GradientDescentOptimizer(learning_rate).minimize(cost,global_st
ep) #梯度下降
32 init = tf.global_variables_initializer()
33 #定义学习参数
34 training_epochs = 20
35 display_step = 2
36 savedir = "log/"
37 saver = tf.train.Saver(tf.global_variables(), max_to_keep=1) #生成saver。
max_to_keep=1, 表示只保留一个检查点文件
38
39 #定义生成 loss 值可视化的函数
40 plotdata = { "batchsize":[], "loss":[] }
41 def moving_average(a, w=10):
42     if len(a) < w:
43         return a[:]
44     return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in
enumerate(a)]
45
46 # (3) 建立会话(session)进行训练
47 with tf.Session() as sess:
48     sess.run(init)
49     kpt = tf.train.latest_checkpoint(savedir)
50     if kpt!=None:
51         saver.restore(sess, kpt)
52
53     #向模型输入数据
54     while global_step.eval()/len(train_X) < training_epochs:
55         step = int( global_step.eval()/len(train_X) )

```

```

56     for (x, y) in zip(train_X, train_Y):
57         sess.run(optimizer, feed_dict={X: x, Y: y})
58
59     # 显示训练中的详细信息
60     if step % display_step == 0:
61         loss = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
62         print ("Epoch:", step+1, "cost=", loss, "W=", sess.run(W), "b=", sess.run(b))
63         if not (loss == "NA"):
64             plotdata["batchsize"].append(global_step.eval())
65             plotdata["loss"].append(loss)
66             saver.save(sess, savedir+"linermodel.cpkt", global_step)
67
68     print (" Finished!")
69     saver.save(sess, savedir+"linermodel.cpkt", global_step)
70     print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}), "W=", sess.run(W), "b=", sess.run(b))
71
72     # 显示模型
73     plt.plot(train_X, train_Y, 'ro', label='Original data')
74     plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
75     plt.legend()
76     plt.show()
77
78     plotdata["avgloss"] = moving_average(plotdata["loss"])
79     plt.figure(1)
80     plt.subplot(211)
81     plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
82     plt.xlabel('Minibatch number')
83     plt.ylabel('Loss')
84     plt.title('Minibatch run vs. Training loss')
85
86     plt.show()

```

本实例中的模型只有一个神经网络节点。由于权重 W 和 b 都是一维的，所以在计算网络正向输出时，直接使用了乘法函数 multiply (X, W)，也可以写成 X\*W。



### 提示：

本实例中的模型非常简单，且输入批次为 1。实际工作中的模型会比这个复杂得多，且每批次都会同时处理多条数据。在计算网络输出时，更多的是用到矩阵相乘。

例如：

```

a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])
with tf.Session() as sess:

```

```
c = tf.matmul(a, b)
print("c", c.eval())
#也可以写成：
c = a @ b
print("c", c.eval())
```

上面代码运行完后，会看到在 log 文件夹下多了几个“linermodel.cpkt-2000”开头的文件。它就是检查点文件。

其中，“2000”表示该文件是运行优化器第 2000 次后生成的检查点文件。

在代码第 34 行，设置了 training\_epochs 的值为“20”，表示将整个数据集迭代 20 次。每迭代一次数据集，需要运行 100 次优化器。



### 提示：

log 文件夹下的几个以“linermodel.cpkt-2000”开头的文件，会在后面 13 章有详细介绍。

扩展名 meta 的文件是图中的网络节点名称文件，可以删掉不影响模型恢复。

这里介绍一个小技巧：在生成模型检查点文件时（代码第 66、69 行），代码可以写成以下样子，让模型不再生成 meta 文件，从而可以减小模型所占的磁盘空间：

```
saver.save(sess, savedir+"linermodel.cpkt", global_step, write_meta_graph=False)
```

## 6.2.5 修改迭代次数，二次训练

将数据集的迭代次数调大到 28（修改代码第 34 行 training\_epochs 的值）。再次运行，输出以下结果：

```
1.13.1
INFO:tensorflow:Restoring parameters from log/linermodel.cpkt-2000
Epoch: 21 cost= 0.088184044 W= [2.0288355] b= [0.00869429]
Epoch: 23 cost= 0.08760502 W= [2.0110996] b= [0.00945178]
Epoch: 25 cost= 0.087475054 W= [2.0058548] b= [0.01136262]
Epoch: 27 cost= 0.08744553 W= [2.004488] b= [0.01188545]
Finished!
cost= 0.08744063 W= [2.0042534] b= [0.01197556]
```

可以看到，输出结果的第 1 行代码直接从以“linermodel.cpkt-2000”开头的文件中读取参数。然后，接着第 20 次迭代继续向下运行（输出结果的第 2 行）。这部分结果对应的代码逻辑如下：

- (1) 查找最近生成的检查点文件（见代码第 49 行）。
- (2) 判断检查点文件是否存在（见代码第 50 行）。
- (3) 如果存在，则将检查点文件的值恢复到张量图中（见代码第 51 行）。

在程序内部是通过张量 global\_step 的载入、载出来记录迭代次数的。

**提示：**

静态图部分是 TensorFlow 的基础操作，但在 TensorFlow 2.x 版本后，已经不再推荐使用。这里也不会讲解得过于详细。如想要系统地了解该部分的知识，建议阅读《深度学习之 TensorFlow——入门、原理与进阶实战》一书的第 4 章。

## 6.3 实例 15：用动态图（eager）训练一个具有保存检查点功能的回归模型

下面实现一个简单的动态图实例。

### 实例描述

假设有这么一组数据集，其  $x$  和  $y$  的对应关系是  $y \approx 2x$ 。

训练模型来学习这些数据集，使模型能够找到其中的规律，即让神经网络自己能够总结出  $y \approx 2x$  这样的公式。

要求使用动态图的方式来实现。同时与实例 13 进行比较，体会动态图和静态图实现时的不同之处。

本实例将内存数据制作成 Dataset 数据集，并在动态图里实现模型。

### 6.3.1 代码实现：启动动态图，生成模拟数据

这部分操作与前面 4.9.1 小节一致。都用 `tf.enable_eager_execution` 函数来启动动态图。在动态图启动之后，便开始生成模拟数据。具体代码如下：

#### 代码 6-2 用动态图训练一个具有保存检查点功能的回归模型

```

01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04 import tensorflow.contrib.eager as tfe
05
06 tf.enable_eager_execution()          # 启动动态图
07 print("TensorFlow 版本: {}".format(tf.VERSION))
08 print("Eager execution: {}".format(tf.executing_eagerly()))
09
10 # 生成模拟数据
11 train_X = np.linspace(-1, 1, 100)
12 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 # y=2x, 但是加入了噪声
13 # 图形显示
14 plt.plot(train_X, train_Y, 'ro', label='Original data')
15 plt.legend()
16 plt.show()
```

生成模拟部分与实例13中的一样，这里不再详述。

### 6.3.2 代码实现：定义动态图的网络结构

定义动态图的网络结构与定义静态图的网络结构有所不同，具体如下：

- 动态图不支持占位符的定义。
- 动态图不能使用优化器的 `minimize` 方法，需要使用 `tfe.implicit_gradients` 方法与优化器的 `apply_gradients` 方法组合（见代码第30、31行）

具体代码如下：

#### 代码6-2 用动态图训练一个具有保存检查点功能的回归模型（续）

```

17 # 定义学习参数
18 W = tf.Variable(tf.random_normal([1]), dtype=tf.float32, name="weight")
19 b = tf.Variable(tf.zeros([1]), dtype=tf.float32, name="bias")
20 global_step = tf.train.get_or_create_global_step()
21
22 def getcost(x,y):# 定义函数，计算 loss 值
23     # 前向结构
24     z = tf.cast(tf.multiply(np.asarray(x,dtype = np.float32), W)+ b,dtype = tf.float32)
25     cost =tf.reduce_mean( tf.square(y - z))# 计算 loss 值
26     return cost
27
28 learning_rate = 0.01
29 # 将随机梯度下降法作为优化器
30 optimizer =
31     tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
32 grad = tfe.implicit_gradients(getcost) # 获得计算梯度的函数

```

代码第31行，用函数 `tfe.implicit_gradients` 生成一个计算梯度的函数——`grad`。在迭代训练的反向传播过程中，`grad` 函数将会被传入优化器的 `apply_gradients` 方法中对模型的参数进行优化，见6.3.4小节。



#### 提示：

函数 `getcost` 的定义（见代码第22行）与使用（见代码第31行），还可以与装饰器的方法合并到一起。例如：

```

@tfe.implicit_gradients
def getcost(x,y):# 定义函数，计算 loss 值
    # 前向结构
    z = tf.cast(tf.multiply(np.asarray(x,dtype = np.float32), W)+ b,dtype = tf.float32)

```

```

cost = tf.reduce_mean(tf.square(y - z))#loss 值
return cost

```

类似该用法的实例参考 6.11 节。

### 6.3.3 代码实现：在动态图中加入保存检查点功能

在动态图中保存检查点有两种方式。

- 用 `tf.train.Saver` 类操作检查点文件：实例化一个对象 `saver`，手动指定参数[W,b]进行保存（见代码第 35 行），并且将会话(session)有关的参数设为 None（见代码第 41 行）。
- 用 `tensorflow.contrib.eager` 模块的 `Saver` 类操作检查点文件：直接实例化一个对象 `saver`，在生成过程中不需要传入会话参数。



#### 提示：

在用 `tf.train.Saver` 类操作检查点文件时，必须手动指定要保存的参数。因为动态图里没有会话和图的概念，所以不支持用 `tf.global_variables` 函数获取所有参数。

具体代码如下：

#### 代码 6-2 用动态图训练一个具有保存检查点功能的回归模型（续）

```

32 #定义 saver, 演示两种方法处理检查点文件
33 savedir = "logeager/"
34 savedirx = "logeagerx/"
35 saver = tf.train.Saver([W,b], max_to_keep=1)#生成 saver。max_to_keep=1 表示
最多只保存一个检查点文件
36 saverx = tfe.Saver([W,b])#生成 saver。max_to_keep=1 表示只保存一个检查点文件
37
38 kpt = tf.train.latest_checkpoint(savedir)    #找到检查点文件
39 kptx = tf.train.latest_checkpoint(savedirx)#找到检查点文件
40 if kpt!=None:
41     saver.restore(None, kpt)#用 tf.train.Saver 的实例化对象加载模型
42     saverx.restore(kptx)      #用 tfe.Saver 的实例化对象加载模型
43
44 training_epochs = 20                  #迭代训练次数
45 display_step = 2

```

在复杂模型中，模型的参数会非常多。用手动指定变量的方式来保存模型（见代码第 35、36 行）会显得过于麻烦。

动态图框架一般会与 `tf.layers` 接口或 `tf.keras` 接口配合使用（在 TensorFlow 2.x 框架中，主要与 `tf.keras` 接口配合使用）。利用这两个接口，可以很容易地将参数放到定义时的 `saver` 对象中。具体可见 6.6 节在动态图中使用 `tf.layers` 接口的实例，以及 9.2 节在动态图中使用 `tf.keras` 接口的实例。

**提示：**

在TensorFlow 2.x版本中，主要推荐用tf.train.Checkpoint方法操作检查点文件。TensorFlow 1.x版本中的tf.train.Saver类未来可能会被去掉。在使用tf.train.Checkpoint方法时，要求必须将网络结构封装成类，否则无法调用，具体用法可以参考9.2节。

### 6.3.4 代码实现：按指定迭代次数进行训练，并可视化结果

迭代训练过程的代码是最容易理解的。它是动态图的真正优势所在，使张量程序像Python中的普通程序一样运行。

在动态图程序中，可以对每个张量的numpy方法进行取值（见代码第66行），不再需要使用run函数与eval方法。

**代码 6-2 用动态图训练一个具有保存检查点功能的回归模型（续）**

```

46 plotdata = { "batchsize":[], "loss":[] } #收集训练参数
47
48 while global_step/len(train_X) < training_epochs: #迭代训练模型
49     step = int( global_step/len(train_X) )
50     for (x, y) in zip(train_X, train_Y):
51         optimizer.apply_gradients(grad(x, y),global_step) #应用梯度
52
53     #显示训练中的详细信息
54     if step % display_step == 0:
55         cost = getcost (x, y) #用于显示
56         print ("Epoch:", step+1, "cost=", cost.numpy(), "W=", W.numpy(), "b=", b.numpy())
57         if not (cost == "NA" ):
58             plotdata["batchsize"].append(global_step.numpy())
59             plotdata["loss"].append(cost.numpy())
60         saver.save(None, savedir+"linermodel.cpkt", global_step)
61         saverx.save(savedirx+"linermodel.cpkt", global_step)
62
63 print (" Finished!")
64 saver.save(None, savedir+"linermodel.cpkt", global_step)
65 saverx.save(savedirx+"linermodel.cpkt", global_step)
66 print ("cost=", getcost (train_X, train_Y).numpy() , "W=", W.numpy(), "b=", b.numpy())
67
68 #显示模型
69 plt.plot(train_X, train_Y, 'ro', label='Original data')
70 plt.plot(train_X, W * train_X + b, label='Fitted line')
71 plt.legend()
72 plt.show()
73
74 def moving_average(a, w=10):#定义生成loss值可视化的函数
75     if len(a) < w:
```

```

76     return a[:,]
77     return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in
78             enumerate(a)]
79 plotdata["avgloss"] = moving_average(plotdata["loss"])
80 plt.figure(1)
81 plt.subplot(211)
82 plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
83 plt.xlabel('Minibatch number')
84 plt.ylabel('Loss')
85 plt.title('Minibatch run vs. Training loss')
86
87 plt.show()

```

### 6.3.5 运行程序，显示结果

代码运行后，输出以下结果：

```

TensorFlow 版本: 1.13.1
Eager execution: True

```

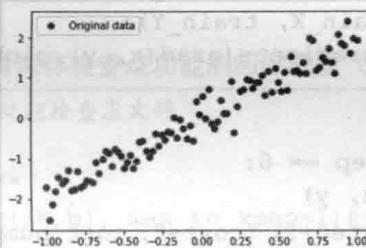


图 6-2 动态图回归模型结果 (a)

```

Epoch: 1 cost= 2.7563627 W= [0.26635304] b= [0.01309205]
Epoch: 3 cost= 0.14655435 W= [1.5330775] b= [0.01505858]
Epoch: 5 cost= 0.0032546197 W= [1.8566017] b= [0.01509316]
Epoch: 7 cost= 0.0006836037 W= [1.9392302] b= [0.01509374]
Epoch: 9 cost= 0.0022461722 W= [1.9603337] b= [0.01509374]
Epoch: 11 cost= 0.0027899994 W= [1.9657234] b= [0.01509374]
Epoch: 13 cost= 0.0029383437 W= [1.9671] b= [0.01509374]
Epoch: 15 cost= 0.0029768397 W= [1.9674516] b= [0.01509374]
Epoch: 17 cost= 0.002986682 W= [1.9675411] b= [0.01509374]
Epoch: 19 cost= 0.0029891713 W= [1.9675636] b= [0.01509374]
Finished!
cost= 0.080912225 W= [1.9675636] b= [0.01509374]

```

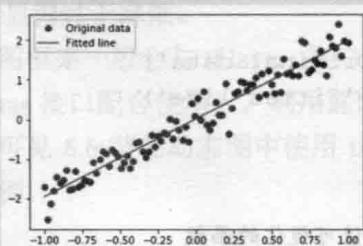


图 6-2 动态图回归模型结果 (b)

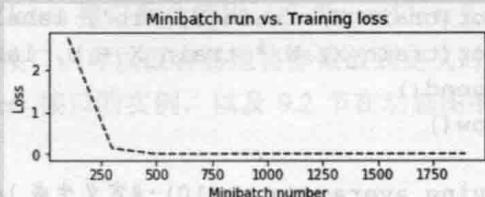


图 6-2 动态图回归模型结果 (c)

图 6-2 (c) 显示的是 loss 值经过移动平均算法的结果（见代码第 75 行）。用移动平均算法可以使生成的曲线更加平滑，便于看出整体趋势。

### 6.3.6 代码实现：用另一种方法计算动态图梯度

在 6.3.2 小节中，介绍了用 `tfe.implicit_gradients` 方法在动态图中进行反向训练。

本节再介绍一种同样很常用的方法——tf.GradientTape。

`tf.GradientTape` 方法可以在反向传播过程中跟踪自动微分（Automatic differentiation）之后的梯度计算工作。

具体代码如下：

代码 6-3 动态图另一种梯度方法

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04 import tensorflow.contrib.eager as tfe
05
06 tf.enable_eager_execution()
07 .....
08 def getcost(x,y): #定义函数，计算 loss 值
09     #前向结构
10     z = tf.cast(tf.multiply(np.asarray(x,dtype = np.float32), W)+ b, dtype = tf.float32)
11     cost =tf.reduce_mean( tf.square(y - z))#loss 值
12     return cost
13
14 def grad( inputs, targets): #封装梯度计算函数
15     with tf.GradientTape() as tape: #用 tf.GradientTape 跟踪梯度计算
16         loss_value = getcost(inputs, targets)
17     return tape.gradient(loss_value,[W,b])
18 .....
19 while global_step/len(train_X) < training_epochs: #迭代训练模型
20     step = int( global_step/len(train_X) )
21     for (x, y) in zip(train_X, train_Y):
22         grads = grad( x, y) #计算梯度
23         optimizer.apply_gradients(zip(grads, [W,b]),
global_step=global_step)
.....
```

相比于代码文件“6-2 中用动态图训练一个具有检查点功能的回归模型”，这里主要改动了两处：

- 在代码 14 行，将损失函数用 `tf.GradientTape` 函数封装起来。
  - 在使用时，需要传入训练参数（见代码第 23 行）。

使用 `tf.GradientTape` 函数可以对梯度做更精细化的控制(可以自由指定需要训练的变量),

而使用 `tfe.implicit_gradients` 函数会使代码变得相对简洁。在 TensorFlow 2.x 中，只保留了 `tf.GradientTape` 函数用于计算梯度。`tfe.implicit_gradients` 函数在 TensorFlow 2.x 中将不再被支持。

在本书的 6.11 节中，还使用了另一种求梯度的方法——`tfe.implicit_value_and_gradients`。该方式同样也是在 `contrib` 模块中的代码。有兴趣的读者可以了解一下。

在真正应用时，可根据实际情况来具体选择。



### 提示：

在用 `tf.GradientTape` 函数计算损失时，要求传入指定的参数。在本节的实例代码中，要计算损失的指定参数（W、b）是预先定义好的。

如果用 TensorFlow 的高级接口构建模型，则参数是在 API 内部定义的，无法直接调试。在这种情况下，可以用 `tfe.EagerVariableStore()` 的方法将动态图的变量保存到全局集合里，然后通过实例化的对象取出变量并传入 `tf.GradientTape` 中。具体操作可以参考 6.3.7 小节。

### 6.3.7 实例 16：在动态图中获取参数变量

动态图的参数变量存放机制与静态图截然不同。

动态图用类似 Python 变量生命周期的机制来存放参数变量，不能像静态图那样通过图的操作获得指定变量。但在训练模型、保存模型等场景中，如何在动态图里获得指定变量呢？这里提供以下两种方法。

- 方法一：将模型封装成类，借助类的实例化对象在内存中的生命周期来管理模型变量，即使用模型的 `variables` 成员变量。这种也是最常用的一种方式（见 6.6 节的 `tf.layers` 接口实例、9.2 节的 `tf.keras` 接口实例）。
- 方法二：用 `tfe.EagerVariableStore()` 方法将动态图的变量保存到全局集合里，然后再通过实例化的对象取出变量。这种方式更加灵活，编程人员不必以类的方式来实现模型。

下面将演示方法二，具体代码如下：

#### 代码 6-4 从动态图中获取变量

```

01 import tensorflow as tf
02 import numpy as np
03 import tensorflow.contrib.eager as tfe
04
05 tf.enable_eager_execution()
06 print("TensorFlow 版本: {}".format(tf.VERSION))
07 print("Eager execution: {}".format(tf.executing_eagerly()))
08
09 #生成模拟数据
10 train_X = np.linspace(-1, 1, 100)
11 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3
12
13 #建立数据集

```

```

14 dataset =
15     tf.data.Dataset.from_tensor_slices( (np.reshape(train_X, [-1,1]),np.reshape(train_Y, [-1,1])) )
16 global_step = tf.train.get_or_create_global_step()
17 container = tfe.EagerVariableStore()          #用于保存动态图变量
18 learning_rate = 0.01
19 #随机梯度下降法作为优化器
20 将 optimizer =
21     tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
22 def getcost(x,y):                         #定义函数，计算 loss 值
23     with container.as_default():#将动态图使用的层包装起来，可以得到变量
24         z = tf.layers.dense(x,1, name="l1")      #前向结构
25         cost =tf.reduce_mean( tf.square(y - z))    #计算 loss 值
26     return cost
27
28 def grad( inputs, targets):#计算梯度函数
29     with tf.GradientTape() as tape:
30         loss_value = getcost(inputs, targets)
31     return tape.gradient(loss_value,container.trainable_variables())
32
33 training_epochs = 20                      #迭代训练次数
34 display_step = 2
35
36 for step,value in enumerate(dataset):       #迭代训练模型
37     grads = grad( value[0], value[1])
38     optimizer.apply_gradients(zip(grads, container.trainable_variables()),
39     global_step=global_step)
40     if step>=training_epochs:
41         break
42
43     #显示训练中的详细信息
44     if step % display_step == 0:
45         cost = getcost (value[0], value[1])
46         print ("Epoch:", step+1, "cost=", cost.numpy())
47
48 print (" Finished!")
49 print ("cost=", cost.numpy() )
50 for i in container.trainable_variables():
51     print(i.name,i.numpy())

```

上面代码的主要流程解读如下：

- (1) 代码第14行，将模拟数据做成了数据集。
- (2) 代码第17行，实例化tfe.EagerVariableStore类，得到container对象。
- (3) 代码第22行，计算损失值函数getcost。在该函数中，通过with container.as\_default作

用域将网络参数保存在 container 对象中。

(4) 代码第 28 行, 计算梯度函数 grad。其中使用了 tf.GradientTape 方法, 并通过 container.trainable\_variables 方法取得需要训练的参数, 然后传入 tape.gradient 中计算梯度。

(5) 代码第 38 行, 再次通过 container.trainable\_variables 方法取得需要训练的参数, 并传入优化器的 apply\_gradients 方法中, 以更新权重参数。

代码运行后, 输出以下结果:

```
TensorFlow 版本: 1.13.1
Eager execution: True
Epoch: 1 cost= 0.11828259153554481
Epoch: 3 cost= 0.09272109443044181
Epoch: 5 cost= 0.07258319799191404
Epoch: 7 cost= 0.05665282399104451
Epoch: 9 cost= 0.04400892987470931
Epoch: 11 cost= 0.033949746009501354
Epoch: 13 cost= 0.025937515234633546
Epoch: 15 cost= 0.01955791804589977
Epoch: 17 cost= 0.014490085910067178
Epoch: 19 cost= 0.010484296911198973
Finished!
cost= 0.010484296911198973
l1/bias:0 [-0.08494885]
l1/kernel:0 [[0.71364929]]
```

在输出结果的倒数第 5 行, 可以看到模型迭代训练了 19 次之后, 损失值 cost 降到了 0.01。

在输出结果的最后两行, 可以看到所训练出来的模型中, 包含有两个权重: “l1/bias:0” 与 “l1/kernel:0”。这表示使用 tf.layers.dense 函数构建的全连接网络模型, 与代码文件“6-3 动态图另一种梯度方法.py”中手动构建的模型具有一样的结构(两个权重)。只不过两者权重名字不同而已(本实例中的权重名称是 l1/bias 和 l1/kernel, 而代码文件“6-3 动态图另一种梯度方法.py”中模型的权重名称是 W 和 b)。



### 提示:

在本实例中, container 对象还可以用 container.variables 方法来获得全部的变量, 以及用 container.non\_trainable\_variables 方法获得不需要训练的变量。

另外, 还需要注意 API 在动态图中的使用。在 6.1.5 小节介绍过, 动态图对 tf.layers 接口的支持比较友好, 但是换为 TF-slim 接口会出问题(详细请见 6.3.8 小节)。

## 6.3.8 小心动态图中的参数陷阱

习惯使用 TF-slim 接口的开发人员, 很容易会将 6.3.7 小节代码中第 24 行用 TF-slim 接口来实现。例如改成以下样子:

```
z = tf.contrib.slim.fully_connected(x, 1) #用 TF-slim 接口实现全连接网络
```

这样的代码整体运行是没有问题的。代码运行后，得到以下的结果：

```
.....
Epoch: 19 cost= 1.4630528048775695
Finished!
cost= 1.4630528048775695
fully_connected/biases:0 [-0.02064418]
.....
fully_connected_30/weights:0 [[-0.8784894]]
fully_connected_4/biases:0 [0.]
.....
fully_connected_9/weights:0 [[-0.83005739]]
```

从结果中会发现两个问题：

- 训练的 loss 值 (cost) 没有收敛 (结果第 2 行显示的值为 1.46)。
- 输出的模型参数并不是两个，而是多个(在输出的结果中，第 5 行之后全是模型参数)。

这表示：在迭代训练中，每调用一次 TF-slim 接口的全连接函数，系统就会重新创建一层全连接网络。最终会生成很多模型参数。

如果尝试使用共享变量的方式解决呢？见下面的代码，将该全连接设为 `tf.AUTO_REUSE`。通过以下代码让其只创建一次：

```
z = tf.contrib.slim.fully_connected(x, 1, reuse=tf.AUTO_REUSE)
```

代码运行后会直接报错。输出以下结果：

```
AttributeError: reuse=True cannot be used without a name_or_scope
```

这是由于共享变量的机制造成的。在动态图中使用了与静态图完全不同的机制，这导致了共享变量失效。可以这样理解：TensorFlow 中的共享变量只在静态图中有效。



### 提示：

本实例是通过打印简单模型输出参数的方法，来排查模型不收敛的问题。在实际环境中，这种问题很难被发现，因为程序可以完美地执行下去，并且模型本来就会有上千个参数。经过多次迭代训练后会出现 loss 值一直不收敛的情况，常常会使人怀疑这是模型自身的结构问题。

尽量在动态图里使用 `tf.layers` 与 `tf.keras` 接口，这样会使开发变得顺畅一些。

## 6.3.9 实例 17：在静态图中使用动态图

在整体训练时，动态图对 loss 值的处理部分显得比静态图烦琐一些。但是在正向处理时，使用动态图确实非常直观、方便。

下面介绍一种在静态图中使用动态图的方法——正向用动态图，反向用静态图。这样可以使程序兼顾二者的优势。

用 `tf.py_function` 函数可以实现在静态图中使用动态图的功能。在 4.7 节的实例中用

tf.py\_function 函数就是为实现这个功能。tf.py\_function 函数可以将正常的 Python 函数封装起来，在动态图中进行张量运算。

修改 6.2 节中的静态图代码，在其中加入动态图部分。具体代码如下：

### 代码 6-5 在静态图中使用动态图

```

01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 .....
06 tf.reset_default_graph()
07
08 def my_py_func(X, W,b):          #将网络中的正向张量图用函数封装起来
09     z = tf.multiply(X, W)+ b
10     print(z)
11     return z
12 .....
13 X = tf.placeholder("float")
14 Y = tf.placeholder("float")
15 #模型参数
16 W = tf.Variable(tf.random_normal([1]), name="weight")
17 b = tf.Variable(tf.zeros([1]), name="bias")
18 #前向结构
19 z = tf.py_function(my_py_func, [X, W,b], tf.float32)    #将静态图改成功动态图
20 global_step = tf.Variable(0, name='global_step', trainable=False)
21 #反向优化
22 cost =tf.reduce_mean( tf.square(Y - z))
23 .....
24     print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}), "W=", sess.run(W), "b=", sess.run(b))
25 #显示模型
26 plt.plot(train_X, train_Y, 'ro', label='Original data')
27 v = sess.run(z, feed_dict={X: train_X})           #再次调用动态图，生成 y 值
28 plt.plot(train_X, v, label='Fitted line')         #将其显示出来
29 plt.legend()
30 plt.show()
```

代码第 19 行，用 tf.py\_function 函数对自定义函数 my\_py\_func 进行了封装。这样，my\_py\_func 函数里的张量便都可以在动态图中运行了。

在 my\_py\_func 函数中，张量 z 可以像 Python 中的数值对象一样直接被使用（见代码第 10 行，可以通过 print 函数将其内部的值直接输出）。在静态图中用动态图的方式可以使模型的调试变得简单。

代码运行后，可以看到以下结果：

```

.....
1.8424727 1.8831174 1.923762 1.9644067 ], shape=(100,), dtype=float32)
Epoch: 33 cost= 0.07197194 W= [2.0119123] b= [-0.04750564]
```

```
tf.Tensor([-2.059418], shape=(1,), dtype=float32)
tf.Tensor([-2.025845], shape=(1,), dtype=float32)
```

上面截取的结果是训练过程中的一个片段。在结果的最后两行输出了 $z$ 的值。可以看到，虽然 $z$ 还是张量，但是已经有值。

代码第10行也可以用`print(z.numpy())`代码来代替，该代码可以直接将 $z$ 的具体值打印出来。

## 6.4 实例18：用估算器框架训练一个回归模型

估算器框架（Estimators API）属于TensorFlow中的一个高级API。由于它对底层代码实现了高度封装，使得开发模型过程变得更加简单。但在带来便捷的同时，也带来了学习成本。本章就来为读者扫清障碍。通过本实例，读者可以掌握估算器的基本开发方法。

### 实例描述

假设有一组数据集，其中 $x$ 和 $y$ 的对应关系为 $y \approx 2x$ 。

本实例就是让神经网络学习这些样本，找到其中的规律，即让神经网络自己能够总结出 $y \approx 2x$ 这样的公式。

要求用估算器框架来实现。

在6.1.4小节中已经介绍了估算器框架的主要组成部分。下面就通过具体实例来介绍如何用估算器框架接口开发模型，以及各个主要部分（输入函数、模型函数、估算器）的代码编写方式。

### 6.4.1 代码实现：生成样本数据集

这部分操作与前面的实例13、实例14中的样本处理方式一致。代码如下：

#### 代码6-6 用估算器框架训练一个回归模型

```
01 import tensorflow as tf
02 import numpy as np
03
04 #在内存中生成模拟数据
05 def GenerateData(datasize = 100):
06     train_X = np.linspace(-1, 1, datasize)           #train_X为-1~1之间连续
    的100个浮点数
07     train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3   #y=2x,
    但是加入了噪声
08     return train_X, train_Y                                #以生成器的方式返回
09
10 train_data = GenerateData()                            #生成原始的训练数据集
11 test_data = GenerateData(20)                          #生成20个测试数据集
12 batch_size=10
13 tf.reset_default_graph()                            #清空图
```

## 6.4.2 代码实现：设置日志级别

可以通过 `tf.logging.set_verbosity` 方法来设置日志级别。

- 当设成 INFO 时，则所有级别高于 INFO 的都可以显示。
- 当设置成其他级别时（例如 ERROR），则只显示级别比 ERROR 高的日志，INFO 将不显示。

### 代码 6-6 用估算器框架训练一个回归模型（续）

---

```
14 tf.logging.set_verbosity(tf.logging.INFO) #能够控制输出信息
```

---

代码第 14 行设置了程序运行时的输出日志级别。在 TensorFlow 中对应的日志级别如下：

```
from tensorflow.python.platform.tf_logging import ERROR
from tensorflow.python.platform.tf_logging import FATAL
from tensorflow.python.platform.tf_logging import INFO
from tensorflow.python.platform.tf_logging import TaskLevelStatusMessage
from tensorflow.python.platform.tf_logging import WARN
...
from tensorflow.python.platform.tf_logging import flush
from tensorflow.python.platform.tf_logging import get_verbosity
from tensorflow.python.platform.tf_logging import info
```

更多的可见代码：

```
Anaconda3\lib\site-packages\tensorflow\tools\api\generator\api\logging\
_init_.py
```

## 6.4.3 代码实现：实现估算器的输入函数

估算器的输入函数实现起来很简单：将原始的数据源转化成为 `tf.data.Dataset` 接口的数据集并返回。

在本实例中，创建了两个输入函数：

- `train_input_fn` 函数用于训练使用，对数据集做了乱序，并且使其可以自我重复使用。
- `eval_input_fn` 函数用于测试及使用模型进行预测，支持不带标签的输入。

具体代码如下：

### 代码 6-6 用估算器框架训练一个回归模型（续）

---

```
15 def train_input_fn(train_data, batch_size): #定义训练数据集输入函数
16     #构造数据集的组成：一个特征输入，一个标签输入
17     dataset =
18         tf.data.Dataset.from_tensor_slices( ( train_data[0],train_data[1] ) )
19     dataset = dataset.shuffle(1000).repeat().batch(batch_size) #将数据集乱
        序、重复、批次组合
20     return dataset #返回数据集
21 #定义在测试或使用模型时数据集的输入函数
22 def eval_input_fn(data,labels, batch_size):
```

---

```

22     #batch 不允许为空
23     assert batch_size is not None, "batch_size must not be None"
24
25     if labels is None:                                #如果是评估，则没有标签
26         inputs = data
27     else:
28         inputs = (data, labels)
29     #构造数据集
30     dataset = tf.data.Dataset.from_tensor_slices(inputs)
31
32     dataset = dataset.batch(batch_size)             #按批次组合
33     return dataset                                  #返回数据集

```

#### 6.4.4 代码实现：定义估算器的模型函数

在定义估算器的模型函数时，函数名可以任意起，但函数的参数与返回值的类型必须是固定的。

##### 1. 估算器模型函数中的固定参数

估算器模型函数中有四个固定的参数。

- **features**: 用于接收输入的样本数据。
- **labels**: 用于接收输入的标签数据。
- **mode**: 指定模型的运行模式，分为 `tf.estimator.ModeKeys.TRAIN`（训练模式）、`tf.estimator.ModeKeys.EVAL`（测试模型）、`tf.estimator.ModeKeys.PREDICT`（使用模型）三个值。
- **params**: 用于传递模型相关的其他参数。

##### 2. 估算器模型函数中的固定返回值

估算器模型函数的返回值有固定要求：必须是一个 `tf.estimator.EstimatorSpec` 类型的对象。该对象的初始化方法如下：

```

def __new__(cls,
          mode,                                     #类实例（属于Python类相关的语法，在类中默认传值）
          predictions=None,                          #使用模式
          loss=None,                                 #返回的预测值节点
          train_op=None,                            #返回的损失函数节点
          eval_metric_ops=None,                      #训练的OP
          export_outputs=None,                        #测试模型时，需要额外输出的信息
          training_chief_hooks=None,                 #导出模型的路径
          training_hooks=None,                       #分布式训练中的主机钩子函数
          scaffold=None,                            #训练中的钩子函数（如果是分布式，将在所有的机器上生效）
          evaluation_hooks=None,                     #使用自定义的操作集合，可以进行自定义初始化、摘要、生成检查点文件等
          prediction_hooks=None):                   #评估模型时的钩子函数

```

在本实例中，用函数 `my_model` 作为模型函数。根据传入的 `mode` 不同，返回不同的

EstimatorSpec 对象，即：

- 如果 mode 等于 ModeKeys.PREDICT 常量，此时模型类型为预测，则返回带有 predictions 的 EstimatorSpec 对象。
- 如果 mode 等于 ModeKeys.EVAL 常量，此时模型类型为评估，则返回带有 loss 的 EstimatorSpec 对象。
- 如果 mode 等于 ModeKeys.TRAIN 常量，此时模型类型为训练，则返回带有 loss 和 train\_op 的 EstimatorSpec 对象。



### 提示：

EstimatorSpec 对象初始化参数中的钩子函数，可以用于监视或保存特定内容，或在图形和会话中进行一些操作。

### 3. 估算器模型函数中的网络结构

在估算器模型函数中定义网络结构的方法，与在正常的静态图中的定义方法几乎一样。估算器框架支持 TensorFlow 中的各种网络模型 API，其中包括：TF-slim、tf.layers、tf.keras 等。

因为估算器本来就是在 tf.layers 接口上构建的，所以在模型中使用 tf.layers 的 API 会更加友好。

下面通过一个最基本的模型来介绍估算器的使用。具体代码如下：

#### 代码 6-6 用估算器框架训练一个回归模型（续）

```

34 def my_model(features, labels, mode, params):#自定义模型函数。参数是固定的：一
    个特征，一个标签
35     #定义网络结构
36     W = tf.Variable(tf.random_normal([1]), name="weight")
37     b = tf.Variable(tf.zeros([1]), name="bias")
38     #前向结构
39     predictions = tf.multiply(tf.cast(features,dtype = tf.float32), W)+ b
40
41     if mode == tf.estimator.ModeKeys.PREDICT: #预测处理
42         return tf.estimator.EstimatorSpec(mode, predictions=predictions)
43
44     #定义损失函数
45     loss = tf.losses.mean_squared_error(labels=labels,
46                                         predictions=predictions)
47
48     meanloss = tf.metrics.mean(loss)#添加评估输出项
49     metrics = {'meanloss':meanloss}
50
51     if mode == tf.estimator.ModeKeys.EVAL: #测试处理
52         return tf.estimator.EstimatorSpec( mode, loss=loss,
53                                         eval_metric_ops=metrics)
54
55     #训练处理

```

```

54     assert mode == tf.estimator.ModeKeys.TRAIN
55     optimizer =
56     tf.train.AdagradOptimizer(learning_rate=params['learning_rate'])
57     train_op = optimizer.minimize(loss,
58                                    global_step=tf.train.get_global_step())
59     return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

```

代码第51行，在返回EstimatorSpec对象时，传入了eval\_metric\_ops参数。eval\_metric\_ops参数会使模型在评估时多显示一个meanloss指标（见代码第48行）。eval\_metric\_ops参数是通过tf.metrics函数创建的，它返回的是一个元组类型对象。

如果需要只显示默认的评估指标，则可以将第51行代码改为：

```
return tf.estimator.EstimatorSpec(mode, loss=loss)
```

即不向EstimatorSpec方法中传入eval\_metric\_ops参数。



#### 提示：

在Anaconda的安装路径中可以找到tf.metrics函数的全部内容，具体路径如下：

\Anaconda3\lib\site-packages\tensorflow\tools\api\generator\api\metrics\\_\_init\_\_.py

在该路径的代码文件中包含准确率、召回率、均值、错误率等一系列常用的评估函数，便于在开发过程中使用。

### 6.4.5 代码实现：通过创建config文件指定硬件的运算资源

在默认情况下，估算器会占满全部显存。如果不想让估算器占满全部显存，则可以用tf.GPUOptions类限制估算器使用的GPU显存。具体做法如下：

#### 代码6-6 用估算器框架训练一个回归模型（续）

```

58 gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333) #构建
59 gpu_options, 防止显存占满
59 session_config=tf.ConfigProto(gpu_options=gpu_options)

```

代码第58行，生成了tf.GPUOptions类的实例化对象gpu\_options。该对象用来控制当前程序，使其只占用系统33.3% GPU显存。

代码第59行，用gpu\_options对象对tf.ConfigProto类进行实例化，生成session\_config对象。session\_config对象就是用于指定硬件运算的变量。



#### 提示：

这种方法也同样适用于会话(session)。一般使用以下方式创建会话(session)：

```
with tf.Session(config=session_config) as sess:
```

#### 1. 估算器占满全部显存所带来的问题

如果不对显存加以限制，一旦当前系统中还有其他程序也在占用GPU，则会报以下错误：

```
InternalError: Blas GEMV launch failed: m=1, n=1
[[Node: linear/linear_model/x/weighted_sum = MatMul[T=DT_FLOAT, transpose_a=false,
transpose_b=false,
_device="/job:localhost/replica:0/task:0/device:GPU:0"] (linear/linear_model/x/Reshape,
linear/linear_model/x/weights/part_0/read/_35)]]
```

为了避免类似问题发生，一般都会对使用的显存加以限制。当多人共享一台服务器进行训练时可以使用该方法。

## 2. 限制显存的其他方法

另外，第 58 行代码还可以写成以下形式：

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.333 # 占用 GPU 33.3% 的显存
```

除指定显存占比外，还可用 allow\_growth 项让 GPU 占用最小显存。例如：

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
```

## 6.4.6 代码实现：定义估算器

估算器的定义主要通过 `tf.estimator.Estimator` 函数来完成。其初始化函数如下：

```
def __init__(self,
            model_fn,           # 类对象实例（属于 Python 类相关的语法，在类中默认传值）
            model_dir=None,     # 自定义的模型目录
            config=None,         # 训练时生成的模型目录
            params=None,         # 配置文件，用于指定运算时的附件条件
            warm_start_from=None): # 传入自定义模型函数中的参数
                           # 热启动的模型目录
```

上述的参数中，热启动(`warm_start_from`)表示从指定目录下的文件参数或 `WarmStartSettings` 对象中，将网络节点的权重恢复到内存中。该功能类似于在二次训练时载入检查点文件(见 6.4.9 节)，常常在对原有模型进行微调时使用。

在代码第 61 行中，用 `tf.estimator.Estimator` 方法生成一个估算器(`estimator`)。该估算器的参数如下：

- 模型函数 `model_fn` 的值为 `my_model` 函数。
- 训练时输出的模型路径是 “`./myestimatormode`”。
- 将学习率 `learning_rate` 放到 `params` 字典里，并将字典 `params` 传入模型。
- 通过 `tf.estimator.RunConfig` 方法生成 `config` 配置参数，并将 `config` 配置参数传入模型。

具体代码如下：

### 代码 6-6 用估算器框架训练一个回归模型（续）

---

```
60 # 构建估算器
61 estimator =
  tf.estimator.Estimator(model_fn=my_model, model_dir='./myestimatormode',
  params={'learning_rate': 0.1},
```

```
config=tf.estimator.RunConfig(session_config=tf.ConfigProto(gpu_options=gpu_options))
62     )
```

在代码第61行中，params里的学习率（learning\_rate）会在my\_model函数中被使用（见代码第55行）。

### 6.4.7 用tf.estimator.RunConfig控制更多的训练细节

在代码第61行中，tf.estimator.Estimator方法中的config参数接收的是一个tf.estimator.RunConfig对象。该对象还有更多关于模型训练的设置项。具体代码如下：

```
def __init__(self,
    model_dir=None,          #指定模型的目录(优先级比estimator的高)
    tf_random_seed=None,      #初始化的随机种子
    save_summary_steps=100,    #保存摘要的频率
    save_checkpoints_steps=_USE_DEFAULT, #生成检查点文件的步数频率
    save_checkpoints_secs=_USE_DEFAULT, #生成检查点文件的时间频率
    session_config=None,       #接受tf.ConfigProto的设置
    keep_checkpoint_max=5,      #保留检查点文件的个数
    keep_checkpoint_every_n_hours=10000, #生成检查点文件的频率
    log_step_count_steps=100,    #在训练过程中,同级loss值的频率
    train_distribute=None):      #通过tf.contrib.distribute.
```

DistributionStrategy指定的一个分布式运算实例

其中，参数save\_checkpoints\_steps和save\_checkpoints\_secs不能同时设置，只能设置一个。

- 如果都没有指定，则默认10分钟保存一次模型。
- 如果都设置为None，则不保存模型。

在本实例中都采用默认的设置。读者可以用具体的参数来调整模型，以熟练掌握各个参数的意义。



#### 提示：

在分布式训练时，keep\_checkpoint\_max可以大一些，否则，一旦超过keep\_checkpoint\_max的检查点文件会被提前回收，将导致其他work在同步估算模型时找不到对应的模型。

### 6.4.8 代码实现：用估算器训练模型

通过调用estimator.train方法可以训练模型。该方法的定义如下：

```
def train(self,
    input_fn,                  #输入函数
    hooks=None,                #钩子函数(优先级比estimator中的钩子的高)
    steps=None,                 #训练的次数
    max_steps=None,             #最大训练次数,为一个累积值
    saving_listeners=None):      #保存的回调函数
```

其中：

- self 是 Python 语法中的类实例对象。
- 输入函数 input\_fn 没有参数。
- hooks 是 SessionRunHook 类型的列表。
- 如果 Steps 为 None，则一直训练，不停止。
- saving\_listeners 是一个 CheckpointSaverListener 类型的列表，可以设置在保存模型过程中的前、中、后环节，对指定的函数进行回调。

在本实例中，传入了指定数据集的输入函数与训练步数。具体代码如下：

#### 代码 6-6 用估算器框架训练一个回归模型（续）

```
63 estimator.train(lambda: train_input_fn(train_data, batch_size), steps=200)
#执行训练 200 次
64
65 tf.logging.info("训练完成.") #输出：训练完成
```

代码运行后，输出以下信息：

```
INFO:tensorflow:Using config: {'_model_dir': './myestimatormode',
'_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoints_steps': None,
'_save_checkpoints_secs': 600, '_session_config': gpu_options {
    per_process_gpu_memory_fraction: 0.333
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000,
'_log_step_count_steps': 100, '_train_distribute': None, '_service': None,
'_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at
0x000002C53AA769B0>, '_task_type': 'worker', '_task_id': 0, '_global_id_in_cluster':
0, '_master': '', '_evaluation_master': '', '_is_chief': True, '_num_ps_replicas': 0,
'_num_worker_replicas': 1}
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow>Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 1 into ./myestimatormode\model.ckpt.
INFO:tensorflow:loss = 2.0265186, step = 0
INFO:tensorflow:global_step/sec: 648.135
INFO:tensorflow:loss = 0.29844713, step = 100 (0.156 sec)
INFO:tensorflow:Saving checkpoints for 200 into ./myestimatormode\model.ckpt.
INFO:tensorflow:Loss for final step: 0.15409622.
INFO:tensorflow:训练完成.
```

在输出信息中，以“INFO”开头的输出信息都可以通过 `tf.logging.set_verbosity` 来设置。最后一行的输出结果是通过 `tf.logging.info` 方法实现的（见代码第 65 行）。

在以“INFO”开头的结果信息中，可以看到第 1 行是估算器的配置项信息。该信息中包含估算器框架在训练时的所有详细参数，可以通过调节这些参数来更好地控制训练过程。

**提示：**

在代码第63行的estimator.train方法中，第1个参数是样本输入函数。该函数使用匿名函数的方法进行了封装。

由于框架支持的输入函数要求没有参数，而自定义的输入函数train\_input\_fn是有参数的，所以这里用一个匿名函数给原有的输入函数train\_input\_fn包上了一层，这样才可以传入estimator.train中。还可以通过偏函数或装饰器技术来实现对输入函数train\_input\_fn的包装。例如：

(1) 偏函数的形式：

```
from functools import partial
estimator.train(input_fn=partial(train_input_fn, train_data=train_data, batch_size=batch_size),
                 steps=200)
```

(2) 装饰器的形式：

```
def wrapperFun(fn): # 定义装饰器函数
    def wrapper(): # 包装函数
        return fn(train_data=train_data, batch_size=batch_size) # 调用原函数
    return wrapper
```

```
@wrapperFun
def train_input_fn2(train_data, batch_size): # 定义训练数据集输入函数
    # 构造数据集的组成：一个特征输入，一个标签输入
    dataset = tf.data.Dataset.from_tensor_slices((train_data[0], train_data[1])) # 将数据集乱序、重复、批次组合
    dataset = dataset.shuffle(1000).repeat().batch(batch_size)
    return dataset # 返回数据集
```

estimator.train(input\_fn=train\_input\_fn2, steps=200)

关于函数封装的更多知识，可以参考《Python带我起飞——入门、进阶、商业实战》一书的6.4节“偏函数”和6.10节“装饰器”。

代码的第63行是将Dataset数据集转化为输入函数。在6.4.10小节中，还会演示一种更简单的方法——直接将Numpy变量转化为输入函数。

#### 6.4.9 代码实现：通过热启动实现模型微调

本小节将通过代码演示热启动的实现，具体步骤如下：

- (1) 重新定义一个估算器estimator2。
- (2) 将事先构造好的warm\_start\_from传入tf.estimator.Estimator方法中。

- (3) 将路径“./myestimatormode”中的检查点文件恢复到估算器 estimator2 中。
- (4) 对估算器 estimator2 进行继续训练，并将训练的模型保存在“./myestimatormode3”中。具体代码如下：

#### 代码 6-6 用估算器框架训练一个回归模型（续）

```

66 #热启动
67 warm_start_from = tf.estimator.WarmStartSettings(
68     ckpt_to_initialize_from='./myestimatormode',
69 )
70 #重新定义带有热启动的估算器
71 estimator2 =
72     tf.estimator.Estimator( model_fn=my_model, model_dir='./myestimatormode3',
73     ,warm_start_from=warm_start_from, params={'learning_rate': 0.1},
74
75 config=tf.estimator.RunConfig(session_config=session_config) )
76 estimator2.train(lambda: train_input_fn(train_data,
77 batch_size), steps=200)

```

代码第 67 行，用 `tf.estimator.WarmStartSettings` 类的实例化来指定热启动文件。模型启动后，将通过 `tf.estimator.WarmStartSettings` 类实例化的对象读取“./myestimatormode”下的模型文件，并为当前模型的权重赋值。

该类的初始化参数有 4 个，具体如下。

- `ckpt_to_initialize_from`: 指定模型文件的路径。系统将会从该路径下加载模型文件，并将其中的值赋给当前模型中的指定权重。
- `vars_to_warm_start`: 指定将模型文件中的哪些变量赋值给当前模型。该值可以是一个张量列表，也可以是指定的张量名称，还可以是一个正则表达式。当该值为正则表达式时，系统会在模型文件里用正则表达式过滤出对应的张量名称。默认值为“\*”。
- `var_name_to_vocab_info`: 该参数是一个字典形式。用于将模型文件恢复到 `tf.estimator.VocabInfo` 类型的张量。默认值都为 `None`。`tf.estimator.VocabInfo` 是对词嵌入的二次封装，支持将原有的词嵌入文件转化为新的词嵌入文件并进行使用。
- `var_name_to_prev_var_name`: 该参数是一个字典形式。当模型文件中的变量符号与当前模型中的变量不同时，则可以用该参数进行转换。默认值为 `None`。

这种方式常用于加载词嵌入文件的场景，即将训练好的词嵌入文件加载到当前模型中指定的词嵌入变量中进行二次训练。有关词嵌入的更多例子，可以参考 7.4.3 小节、8.3.2 小节的实例。

代码运行后，生成以下结果（实际输出中并没有序号）：

```

1. INFO:tensorflow:Using      config:      {'_model_dir':      './myestimatormode',
'_tf_random_seed': None,
2. .....
3. INFO:tensorflow:Saving checkpoints for 200 into ./myestimatormode\model.ckpt.
4. INFO:tensorflow:Loss for final step: 0.14718035.
5. INFO:tensorflow:训练完成。

```

```

6.   INFO:tensorflow:Using config: {'_model_dir': './myestimatormode3',
7.   '_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoints_steps':
None, '_save_checkpoints_secs': 600, '_session_config': gpu_options {
8.     per_process_gpu_memory_fraction: 0.333
9.   }
10.  .....
11.  INFO:tensorflow:Warm-starting with WarmStartSettings:
12.  WarmStartSettings(ckpt_to_initialize_from='./myestimatormode',
13.  vars_to_warm_start='.*', var_name_to_vocab_info={}, var_name_to_prev_var_name={})
14.  INFO:tensorflow:Warm-starting from: ('./myestimatormode',)
15.  INFO:tensorflow:Warm-starting variable: weight; prev_var_name: Unchanged
16.  INFO:tensorflow:Initialize variable weight:0 from checkpoint ./myestimatormode
with weight
17.  .....
18.  INFO:tensorflow:Saving checkpoints for 200 into ./myestimatormode3\model.ckpt.
19.  INFO:tensorflow:Loss for final step: 0.08332317.

```

下面介绍输出结果。

- 第3行，显示了模型的保存路径是“./myestimatormode\model.ckpt”。
- 第5行，显示了估算器 estimator 的训练结束。
- 从第6行开始，是估算器 estimator2 的创建。在第2个省略号的下一行，可以看到屏幕输出了“INFO:tensorflow:Warm-starting”，这表示 estimator2 实现了热启动模式，正在从“./myestimatormode\model.ckpt”中恢复参数。
- 第16行，显示模型恢复完参数后开始继续训练。
- 第18行，显示估算器 estimator2 将训练的结果保存到“./myestimatormode3\model.ckpt”下，完成了微调模型的操作。



### 提示：

这里介绍一个使用 tf.estimator.WarmStartSettings 类时的调试技巧。

由于 tf.estimator 属于高集成框架，所以，如果使用了带有正则表达式的 tf.estimator.WarmStartSettings 类，则一旦代码出错会非常难于调试。

如果在估算器的模型代码中引入了 warm\_starting\_util 模块，则可以对 WarmStartSettings 类的正则表达式进行独立调试，以确保热启动环节正常运行，从而降低 tf.estimator 框架的复杂度。

具体代码如下：

```

import tensorflow as tf
from tensorflow.python.training import warm_starting_util #引入 warm_starting_util 模块
with tf.Graph().as_default() as g: #定义静态图

```

```

with tf.Session(graph=g) as sess:          #建立会话
    W = tf.Variable(tf.random_normal([1]), name="weight") #定义热启动目标权重
    #测试热启动功能
    warm_starting_util.warm_start('./myestimatormode', vars_to_warm_start='.*weight.*')
    sess.run(tf.global_variables_initializer())
    print(W.eval())                         #输出热启动得到的变量结果

```

代码运行后，程序成功将模型文件中的变量 W 恢复到当前模型并输出。运行结果如下：

```

INFO:tensorflow:Warm-starting from: ('./myestimatormode',)
INFO:tensorflow:Warm-starting variable: weight; prev_var_name: Unchanged
[2.146502]

```

#### 6.4.10 代码实现：测试估算器模型

测试的代码与训练的代码非常相似。直接调用 estimator 的 evaluate 方法，并传入输入函数即可。

在本实例中，使用了估算器的另一个输入函数的方法——tf.estimator.inputs.numpy\_input\_fn。该方法直接可以把 Numpy 变量的数据包装成一个输入函数返回。

具体代码如下：

##### 代码 6-6 用估算器框架训练一个回归模型（续）

---

```

74 test_input_fn = tf.estimator.inputs.numpy_input_fn(test_data[0], test_data
[1], batch_size=1, shuffle=False)
75 train_metrics = estimator.evaluate(input_fn=test_input_fn)
76 print("train_metrics", train_metrics)

```

---

代码第 74 行，将 Numpy 类型变量制作成估算器的输入函数。与该方法类似，还可以用 tf.estimator.inputs.pandas\_input\_fn 方法将 Pandas 类型变量制作成估算器的输入函数（见 7.8 节实例）。

代码运行后，输出以下结果：

```

.....
INFO:tensorflow:Saving dict for global step 200: global_step = 200, loss = 0.08943534,
meanloss = 0.08943534
train_metrics {'loss': 0.08943534, 'meanloss': 0.08943534, 'global_step': 200}

```

在输出结果的最后一行可以看到“meanloss”这一项，该信息就是代码第 48 行中添加的输出信息。

#### 6.4.11 代码实现：使用估算器模型

调用 estimator 的 predict 方法，分别将测试数据集和手动生成的数据传入模型中进行预测。

- 在使用测试数据集时，调用输入函数 eval\_input\_fn（见代码第21行），并传入值为 None 的标签。
- 在使用手动生成的数据时，用函数 tf.estimator.inputs.numpy\_input\_fn 生成输入函数 predict\_input\_fn，并将输入函数 predict\_input\_fn 传入估算器的 predict 方法。

具体代码如下：

#### 代码 6-6 用估算器框架训练一个回归模型（续）

```

77 predictions = estimator.predict(input_fn=lambda:
    eval_input_fn(test_data[0],None,batch_size))
78 print("predictions",list(predictions))
79 #定义输入数据
80 new_samples = np.array([ 6.4, 3.2, 4.5, 1.5], dtype=np.float32)
81 predict_input_fn =
    tf.estimator.inputs.numpy_input_fn( new_samples,num_epochs=1,
    batch_size=1,shuffle=False)
82 predictions = list(estimator.predict(input_fn=predict_input_fn))
83 print("输入, 结果: {} {}".format(new_samples,predictions))

```

函数 estimator.predict 的返回值是一个生成器类型。需要将其转化为列表才能打印出来（见代码第82行）。

代码运行后，输出以下结果：

```

.....
INFO:tensorflow:Restoring parameters from ./myestimatormode\model.ckpt-200
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
predictions [-1.8394374, -1.6450617, -1.4506862, -1.2563106, -1.061935, -0.8675593,
-0.6731837, -0.4788081, -0.28443247, -0.09005685, 0.10431877, 0.29869437, 0.49307,
0.68744564, 0.8818213, 1.0761969, 1.2705725, 1.4649482, 1.6593237, 1.8536993]
.....
INFO:tensorflow:Restoring parameters from ./myestimatormode\model.ckpt-200
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
输入, 结果: [6.4 3.2 4.5 1.5] [11.825169, 5.91615, 8.316689, 2.7769835]

```

从输出结果中可以看出，两种数据都有正常的输出。

如果是在生产环境中，还可以将估算器的模型保存成冻结图文件，通过 TF Serving 模块来部署。见 13.2 节的实例。

#### 6.4.12 实例 19：为估算器添加日志钩子函数

将代码文件“6-6 用估算器框架训练一个回归模型.py”复制一份，并在其内部添加日志钩子函数，将模型中的 loss 值按照指定步数输出。

## 1. 在模型中添加张量

在模型函数 my\_model 中，用 tf.identity 函数复制张量 loss，并将新的张量命名为“loss”。具体代码如下：

代码 6-7 为估算器添加钩子

```

01 def my_model(features, labels, mode, params):#自定义模型函数：参数是固定的。一个特征，一个标签
02     .....
03     return tf.estimator.EstimatorSpec(mode, predictions=predictions)
04
05     #定义损失函数
06     loss = tf.losses.mean_squared_error(labels=labels,
07     predictions=predictions)
08     lossout = tf.identity(loss, name="loss")           #复制张量用于显示
09     meanloss = tf.metrics.mean(loss)                 #添加评估输出项
10     .....
11     return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

```

## 2. 定义钩子函数，并加入训练中

在调用训练模型方法 estimator.train 之前，用函数 tf.train.LoggingTensorHook 定义好钩子函数，并将生成的钩子函数 logging\_hook 放入 estimator.train 方法中。

具体代码如下：

代码 6-7 为估算器添加钩子（续）

```

.....
12 tensors_to_log = {"钩子函数输出": "loss"}          #定义要输出的内容
13 logging_hook = tf.train.LoggingTensorHook(tensors=tensors_to_log,
14                                           every_n_iter=1)
15 estimator.train(lambda: train_input_fn(train_data, batch_size), steps=200,
16                  hooks=[logging_hook])
17 tf.logging.info("训练完成。")#输出训练完成

```

代码第 13 行用 tf.train.LoggingTensorHook 函数生成了钩子函数 logging\_hook。该函数中的参数 every\_n\_iter 表示，在迭代训练中每训练 every\_n\_iter 次就调用一次钩子函数，输出参数 tensors 所指定的信息。

代码执行后输出如下结果：

```

.....
INFO:tensorflow:钩子函数输出 = 0.0732526 (0.004 sec)
INFO:tensorflow:钩子函数输出 = 0.09113709 (0.004 sec)
INFO:tensorflow:Saving checkpoints for 4200 into ./estimator_hook\model.ckpt.
INFO:tensorflow:Loss for final step: 0.09113709.
INFO:tensorflow:训练完成。

```

从结果中可以看出，程序每迭代训练一次，输出一次钩子信息。

在本书9.4节还会介绍一个在估算器中用hook输出模型节点的例子。另外在随书配套资源中，还有一个关于自定义hook配合tf.train.MonitoredSession使用的例子，具体请见代码文件“6-8自定义hook.py”。

## 6.5 实例20：将估算器代码改写成静态图代码

对于使用者来说，估算器框架在带来便捷开发的同时也带来了不方便性。如果要对模型做更为细节的调整和改进，则优先使用静态图或动态图框架。

本实例将估算器代码改写成静态图代码。

### 实例描述

在6.4节中，有一个用估算器实现的模型，能够实现 $y \approx 2x$ 的关系。需要先将该估算器模型转成静态图模型，然后重用估算器模型训练所生成的检查点文件。

本实例参照6.4节代码进行开发，将估算器代码改写成静态图代码。

需要以下几个步骤。

- (1) 复制网络结构：将6.4节代码中my\_model函数中的网络结构重新复制一份，作为静态图的网络结构。
- (2) 重用输入函数：将输入函数生成的数据集作为静态图的输入数据源。
- (3) 创建会话恢复模型：在会话里载入检查点文件。
- (4) 继续训练。

### 6.5.1 代码实现：复制网络结构

作为程序的开始部分，在复制网络结构之前需要引入模块，并把模拟生成数据集函数一起移植过来。

在复制网络结构时，还需要额外处理几个地方。

- 定义输入占位符(features、labels)：在6.4节的my\_model函数中，features、labels是估算器传入的迭代器变量，在静态图中已经不再适合，所以需要手动定义输入节点。
- 定义全局计步器(global\_step)：估算器框架会在内部生成一个global\_step，但是普通的静态图模型并不会自动创建global\_step，所以需要手动定义一个global\_step。
- 定义保存文件对象(saver)：在估算器框架中，saver也是内置的。在静态图中，需要重新创建。

具体代码如下：

#### 代码6-9 将估算器模型转为静态图模型

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
```

```

05 #在内存中生成模拟数据
06 def GenerateData(datasize = 100):
07     train_X = np.linspace(-1, 1, datasize) #train_X 是-1~1 之间连续的 100 个
08     浮点数
09     train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3
10     return train_X, train_Y #以生成器的方式返回
11
12
13 batch_size=10
14
15 def train_input_fn(train_data, batch_size): #定义训练数据集输入函数
16     #构造数据集的组成：一个特征输入，一个标签输入
17     dataset =
18         tf.data.Dataset.from_tensor_slices( ( train_data[0],train_data[1]) )
19     dataset = dataset.shuffle(1000).repeat().batch(batch_size) #将数据集乱序、
20     重复、批次组合
21     return dataset #返回数据集
22
23 #定义生成 loss 值可视化的函数
24 plotdata = { "batchsize":[], "loss":[] }
25 def moving_average(a, w=10):
26     if len(a) < w:
27         return a[:]
28     return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in
29     enumerate(a)]
30
31 tf.reset_default_graph()
32
33 features = tf.placeholder("float",[None]) #重新定义占位符
34 labels = tf.placeholder("float",[None])
35
36 #其他网络结构不变
37 W = tf.Variable(tf.random_normal([1]), name="weight")
38 b = tf.Variable(tf.zeros([1]), name="bias")
39 predictions = tf.multiply(tf.cast(features,dtype = tf.float32), W)+ b#前向
40
41 loss = tf.losses.mean_squared_error(labels=labels,
42 predictions=predictions) #定义损失函数
43
44 global_step = tf.train.get_or_create_global_step() #重新定义 global_step
45
46 optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
47 train_op = optimizer.minimize(loss, global_step=global_step)
48
49 saver = tf.train.Saver(tf.global_variables(), max_to_keep=1) #重新定义 saver

```

代码第39行，用函数`tf.train.get_or_create_global_step`生成张量`global_step`。这样做的好处是：不用再考虑自定义的`global_step`与估算器中的`global_step`类型匹配问题。



### 提示：

定义保存文件对象（`saver`）必须放在网络定义的最后进行创建，否则在其后面定义的变量将不会被`saver`对象保存到检查点文件中。

原因是：在生成`saver`对象时，系统会用`tf.global_variables`函数获得当前图中的所有变量，并将这些变量保存到`saver`对象的内部空间中，用于保存或恢复。如果生成`saver`对象的代码在定义网络结构的代码之前，则`tf.global_variables`函数将无法获得当前图中定义的变量。

## 6.5.2 代码实现：重用输入函数

直接使用在6.4节中实现的输入函数`train_input_fn`，该函数将返回一个`Dataset`类型的数据集。从该数据集中取出张量元素，用于输入模型。

具体实现见以下代码：

### 代码 6-9 将估算器模型转为静态图模型（续）

```
45 # 定义学习参数
46 training_epochs = 500 # 设置迭代次数为 500
47 display_step = 2
48
49 dataset = train_input_fn(train_data, batch_size) # 复用输入函数 train_input_fn
50 one_element = dataset.make_one_shot_iterator().get_next() # 获得输入数据的张量
```

## 6.5.3 代码实现：创建会话恢复模型

估算器生成的检查点文件，与一般静态图的模型文件完全一致。只要在载入模型值前保证当前图的结构与模型结构一致即可（6.5.1小节做的事情）。具体见以下代码：

### 代码 6-9 将估算器模型转为静态图模型（续）

```
51 with tf.Session() as sess:
52
53     # 恢复估算器的检查点
54     savedir = "myestimatormode/"
55     kpt = tf.train.latest_checkpoint(savedir)          # 找到检查点文件
56     print("kpt:", kpt)
57     saver.restore(sess, kpt)                          # 恢复检查点数据
```

## 6.5.4 代码实现：继续训练

该部分代码没有新知识点。具体代码如下：

## 代码 6-9 将估算器模型转为静态图模型（续）

```

58 # 向模型输入数据
59     while global_step.eval() < training_epochs:
60         step = global_step.eval()
61         x, y = sess.run(one_element)
62
63         sess.run(train_op, feed_dict={features: x, labels: y})
64
65         # 显示训练中的详细信息
66         if step % display_step == 0:
67             vloss = sess.run(loss, feed_dict={features: x, labels: y})
68             print ("Epoch:", step+1, "cost=", vloss)
69             if not (vloss == "NA"):
70                 plotdata["batchsize"].append(global_step.eval())
71                 plotdata["loss"].append(vloss)
72             saver.save(sess, savedir+"linermodel.cpkt", global_step)
73
74         print (" Finished!")
75         saver.save(sess, savedir+"linermodel.cpkt", global_step)
76
77         print ("cost=", sess.run(loss, feed_dict={features: x, labels: y}))
78
79         plotdata["avgloss"] = moving_average(plotdata["loss"])
80         plt.figure(1)
81         plt.subplot(211)
82         plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
83         plt.xlabel('Minibatch number')
84         plt.ylabel('Loss')
85         plt.title('Minibatch run vs. Training loss')
86
87         plt.show()

```

运行代码后，输出以下结果：

```

.....
Epoch: 483 cost= 0.08857741
Epoch: 485 cost= 0.07745837
Epoch: 487 cost= 0.07305251
Epoch: 489 cost= 0.14077939
Epoch: 491 cost= 0.035170306
Epoch: 493 cost= 0.025990102
Epoch: 495 cost= 0.07111463
Epoch: 497 cost= 0.08413558
Epoch: 499 cost= 0.074357346
Finished!
cost= 0.07475543

```

显示的损失值曲线如图 6-3 所示。

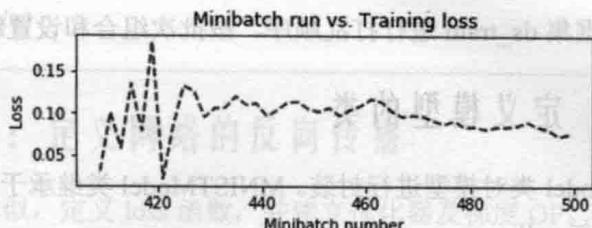


图 6-3 静态图对估算器生成的模型进行二次训练

从结果和损失曲线可以看出，程序运行正常。



### 练习题：

在TensorFlow 2.x 版本之后，动态图框架会变得更加常用。读者可以根据本节的方法，结合动态图的特性（见 6.3 节），自己尝试将估算器代码改写成动态图代码。

## 6.6 实例 21：用 tf.layers API 在动态图上识别手写数字

本实用一个卷积网络在 MNIST 数据集上进行识别任务。通过该实例，演示如何用 tf.layers API 构建模型，并在动态图中进行训练。

### 实例描述

有一组手写数字图片。要求用 tf.layers API 在动态图上搭建模型，将其识别出来。

### 6.6.1 代码实现：启动动态图并加载手写图片数据集

本例加载TFDS 模块中集成好的 MNIST 数据集。该数据集常用于验证模型的功能性实验中。用 tf.enable\_eager\_execution 函数启动动态图，并加载 MNIST 数据集。具体代码如下：

#### 代码 6-10 tf\_layer 模型

```

01 import tensorflow as tf
02 import tensorflow.contrib.eager as tfe
03 tf.enable_eager_execution()
04 print("TensorFlow 版本: {}".format(tf.VERSION))
05 import tensorflow_datasets as tfds
06 import numpy as np
07 #加载训练和验证数据集
08 ds_train, ds_test = tfds.load(name="mnist", split=["train", "test"])
09 ds_train =
10     ds_train.shuffle(1000).batch(10).prefetch(tf.data.experimental.AUTOTUNE)

```

代码第 8 行，调用 tfds.load 方法加载 MNIST 数据集。该方法返回的两个变量 ds\_train 与 ds\_test 都属于 DatasetV1Adapter 类型。DatasetV1Adapter 类型的数据集的使用方式与 tf.data.Dataset 接口的数据集的使用方式非常相似。

代码第 9 行，对数据集 `ds_train` 进行打乱顺序、按批次组合和设置缓存操作。

## 6.6.2 代码实现：定义模型的类

下面定义 `MNISTModel` 类对模型进行封装。`MNISTModel` 类继承于 `tf.layers.Layer` 类。其中有两个方法——`_init_` 与 `call`。

- `_init_` 用于定义网络的各个操作层。本实例中所用到的卷积网络、全连接网络都是用 `tf.layers` 实现的，其用法与 TF-slim 接口非常相似。
- `call` 用于将网络中的各层链接起来，形成正向运算的神经网络。

整个网络结构是：卷积操作+最大池化+卷积操作+最大池化+全连接+dropout 方法+全连接。其中，卷积和池化部分在第 8 章还会深入探讨。

全连接是最基础的神经网络模型之一，该网络的结构是将所有的下层节点与每一个上层节点全部连在一起。

`dropout` 是一种改善过拟合的方法。通过随机丢弃部分网络节点来忽略数据集中的小概率样本。

具体代码如下：

代码 6-10 `tf_layer` 模型（续）

```

10 class MNISTModel(tf.layers.Layer):                                # 定义模型类
11     def __init__(self, name):
12         super(MNISTModel, self).__init__(name=name)
13
14         self._input_shape = [-1, 28, 28, 1]                         # 定义输入形状
15         # 定义卷积层
16         self.conv1 = tf.layers.Conv2D(32, 5, activation=tf.nn.relu)
17         # 定义卷积层
18         self.conv2 = tf.layers.Conv2D(64, 5, activation=tf.nn.relu)
19         # 定义全连接层
20         self.fc1 = tf.layers.Dense(1024, activation=tf.nn.relu)
21         self.fc2 = tf.layers.Dense(10)
22         self.dropout = tf.layers.Dropout(0.5) # 定义 dropout 层
23         # 定义池化层
24         self.max_pool2d = tf.layers.MaxPooling2D(
25             (2, 2), (2, 2), padding='SAME')
26
27     def call(self, inputs, training):                                # 定义 call 方法
28         x = tf.reshape(inputs, self._input_shape)                   # 将网络连接起来
29         x = self.conv1(x)
30         x = self.max_pool2d(x)
31         x = self.conv2(x)
32         x = self.max_pool2d(x)
33         x = tf.keras.layers.Flatten()(x)
34         x = self.fc1(x)
35         if training:
36             x = self.dropout(x)

```

```

37     x = self.fc2(x)
38     return x

```

### 6.6.3 代码实现：定义网络的反向传播

该部分与6.3节类似，定义loss函数，并建立优化器及梯度OP。具体代码如下：

代码6-10 tf\_layer模型（续）

```

39 def loss(model, inputs, labels):
40     predictions = model(inputs, training=True)
41
42     cost =
43         tf.nn.sparse_softmax_cross_entropy_with_logits( logits=predictions,
44             labels=labels )
45     return tf.reduce_mean( cost )
46 #训练
47 optimizer = tf.train.AdamOptimizer(learning_rate=1e-4)
48 grad = tfe.implicit_gradients(loss)

```

### 6.6.4 代码实现：训练模型

该部分与6.3节类似：定义loss函数，并建立优化器及梯度OP。具体代码如下：

代码6-10 tf\_layer模型（续）

```

47 model = MNISTModel("net")           #实例化模型
48 global_step = tf.train.get_or_create_global_step()    #按照指定次数迭代数据集
49 for epoch in range(1):
50     for i,data in enumerate(ds_train):
51         inputs, targets = tf.cast( data["image"],tf.float32), data["label"]
52         optimizer.apply_gradients(grad( model,inputs, targets),
53                                     global_step=global_step)
54         if i % 100 == 0:
55             print("Step %d: Loss on training set : %f" %
56                   (i, loss(model,inputs, targets).numpy()))
57             #获取要保存的变量
58             all_variables = ( model.variables + optimizer.variables() +
59             [global_step])
60             tfe.Saver(all_variables).save(      #生成检查点文件
61                 "./tfelog/linermodel.cpkt", global_step=global_step)
62 ds = tfds.as_numpy(ds_test.batch(100))
63 onetestdata = next(ds)
64 print("Loss on test set: %f" %
65       loss( model,onetestdata["image"].astype(np.float32),
66             onetestdata["label"]).numpy())

```

代码第57行，手动将要保存的文件一起传入tfe.Saver进行保存。这是动态图接口使用起来相对不方便的地方。它并不能自动将全局的变量都搜集起来。

代码运行后，输出以下结果：

```
TensorFlow 版本: 1.13.1
Step 0: Loss on training set : 2.252767
.....
Step 5600: Loss on training set : 0.002125
Loss on test set: 0.055677
```

## 6.7 实例 22：用 tf.keras API 训练一个回归模型

本实例将开发一个简单的回归模型，以此来演示 tf.keras API 的基本使用方法。

### 实例描述

用 tf.keras API 开发模型，对一组数据进行拟合，找出  $y \approx 2x$  的对应关系。

tf.keras API 是 TensorFlow 中一个集成了 Keras 框架语法的高级 API。用 tf.keras API 开发神经网络模型的过程，与在 Keras 框架中开发神经网络的过程非常相似。

### 6.7.1 代码实现：用 model 类搭建模型

在 tf.keras API 中，搭建模型主要有两种：

- 用基础的 model 类来搭建模型。
- 用更高级的 Sequential 类来搭建模型。

本小节将通过实例代码来演示用 model 类搭建模型的方法。

#### 1. 搭建模型的最基本步骤

首先演示一下用 tf.keras 接口搭建模型的最基本步骤。具体代码如下：

代码 6-11 keras 回归模型

```
01 import tensorflow as tf
02 import numpy as np
03 import os
04
05 #在内存中生成模拟数据
06 def GenerateData(datasize = 100 ):
07     train_X = np.linspace(-1, 1, datasize) #train_X是-1~1之间连续的 100 个
浮点数
08     train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3
09     return train_X, train_Y #以生成器的方式返回
10
11 train_data = GenerateData()
12
13 #直接用 model 定义网络
14 inputs = tf.keras.Input(shape=(1,)) #构建输入层
15 outputs= tf.keras.layers.Dense(1)(inputs) #构建全连接层
```

```
16 model = tf.keras.Model(inputs=inputs, outputs=outputs) #构建模型
```

代码第13行之前是创建数据集的操作。

代码第14~16行用tf.keras接口搭建模型，其步骤如下。

(1) 构建输入层：与TensorFlow框架中的占位符类似，用于输入数据。在指定形状(shape)时，不需要指定批次维度。

(2) 构建全连接层：使用了Dense类，后面的第1个括号是该类的实例化。这里传入了1，代表一个输出节点。第2个括号代表对该类实例化对象的函数调用，将输入层传入全连接网络，并生成outputs网络节点。

(3) 创建模型：用于在图中生成网络的正向模型。只需指定输入的张量节点和输出的张量节点即可。

## 2. 搭建多层模型

仿照代码第15行的写法构建两个全连接层，并将它们依次连起来，实现多层模型的搭建。具体代码如下。

**代码6-11 keras回归模型（续）**

```
17 x = tf.keras.layers.Dense(1, activation='tanh')(inputs)      #第1层全连接
18 outputs_2 = tf.keras.layers.Dense(1)(x)                      #第2层全连接
19 model_2 = tf.keras.Model(inputs=inputs, outputs=outputs_2) #定义模型
```

代码第18行中定义了第2层全连接，并将第1层全连接的输出（见代码第17行）作为输入，完成了二层网络的定义。

对于其他类型的网络（卷积、循环等）可以参考Keras框架的教程。原有的Keras框架语法用keras.layers.Dense类，而在tf.keras接口中需要用tf.keras.layers.Dense类。

## 3. 继承Model类，进行搭建网络

除采用对tf.keras.Model类进行实例化的方式来构建模型外，还可以通过定义tf.keras.Model类的子类方式来构建模型。具体操作如下：

(1) 定义一个子类继承tf.keras.Model类。

(2) 在子类的\_\_init\_\_方法中，对模型中的各层网络进行单独定义。

(3) 在子类的call方法中，将定义好的各层网络连起来。

该过程的具体代码演示可以参考9.2节的例子代码。

## 6.7.2 代码实现：用sequential类搭建模型

下面开始介绍用更高级的Sequential类来搭建模型。

用Sequential类搭建模型更为灵活：可以指定输入层的维度、形状。具体步骤如下：

### 1. 用Sequential类搭建模型的基本步骤

使用Model类的方式是“先搭建网络，后定义模型”。而用Sequential类搭建模型的方式与直接使用Model类的方式正相反，是“先定义模型，后搭建网络”。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```
20 model_3 = tf.keras.models.Sequential() # 定义模型对象
21 model_3.add(tf.keras.layers.Dense(1, input_shape=(1,))) # 添加一层全连接
22 model_3.add(tf.keras.layers.Dense(units = 1)) # 再添加一层全连接
```

在代码第 20 行中，定义了一个模型对象 model\_3。然后，用该模型对象的 add 方法将神经网络逐层搭建起来。

在用 Sequential 类定义网络模型过程中，不需要额外定义输入层，直接在第 1 层指定输入的形状即可。后续的神经网络层会自动根据上一层的输出设置自己的输入形状层。

**提示：**

如果网络层数较多，则构建时需要写很多个 model\_3.add，显然非常不方便。还可以用以下的简单方法将所有的网络层放到数组里传入：

```
model_3=tf.keras.models.Sequential( [      tf.keras.layers.Dense(1, input_shape=(1,)),
tf.keras.layers.Dense(units = 1)      ]
)
```

**2. 通过带指定批次的 input 形状来搭建模型**

在模型的第 1 层，还可以用 batch\_input\_shape 参数来描述输入层。在 batch\_input\_shape 参数的赋值过程中所指定的形状要包含批次信息，这与指定占位符形状的方式完全一致。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```
23 model_4 = tf.keras.models.Sequential() # 定义模型
24 model_4.add(tf.keras.layers.Dense(1, batch_input_shape=(None, 1))) # 添加全
连接网络层时，为输入层指定带批次的形状
```

代码第 24 行用网络模型 model\_4 的 add 方法将全连接网络加入。

**3. 通过指定 input 的维度来搭建模型**

还可以使用更为简化的方式来构建模型的第 1 层：直接将输入张量的维度数量传入 input\_dim 参数。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```
25 model_5 = tf.keras.models.Sequential() # 定义模型
26 model_5.add(tf.keras.layers.Dense( 1, input_dim = 1)) # 指定输入维度
```

代码第 28 行，用网络模型 model\_5 的 add 方法将链接网络加入进去。

**4. 用默认输入来搭建模型**

如果在构建模型第 1 层时没有对模型的输入进行设置，则系统将用默认的输入搭建模型。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```

27 model_6 = tf.keras.models.Sequential()          # 定义模型
28 model_6.add(tf.keras.layers.Dense(1))           # 用默认输入添加层
29 print(model_6.weights)                         # 打印模型权重参数
30 model_6.build((None, 1))                      # 指定输入，开始生成模型
31 print(model_6.weights)                         # 打印模型权重参数

```

在代码第 28 行中，直接添加了一个网络层，却没有指定输入，这样也是可以的。但是模型并不会马上构建，只有通过模型的 build 方法或 fit 方法才会触发构建模型的事件。

这里用 build 方法构建网络。在构建过程中，需要为模型指定输入形状。

如果使用 fit 方法，则不需要为模型指定输入形状，因为 fit 方法会通过传入的输入数据来自动识别出输入的形状，然后构建网络。fit 方法可以同时完成网络的构建与训练，一般用在训练模型场景中（见 6.7.4 小节）。

运行上面这段代码后，会输出以下结果：

```

[]
[<tf.Variable 'dense_68/kernel:0' shape=(1, 1) dtype=float32>, <tf.Variable
'dense_68/bias:0' shape=(1,) dtype=float32>]

```

输出结果中第 1 行是空数组，表示模型并没有构建网络节点。

第 2 行是模型调用 build 之后的权重输出。可以看到，显示了具体的张量，这表示模型已经被构建。

**提示：**

`tf.keras.Sequential` 方式虽然比较方便，但它仅仅适用于按顺序堆叠的模型，无法表示复杂的模型，例如多输入、多输出、带有共享层的模型、非序列的数据流模型（残差连接）等。

### 6.7.3 代码实现：搭建反向传播的模型

搭建模型的反向传播过程只需要 1 行代码，即直接调用 Model 类的 compile 方法。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```

32 model.compile(loss = 'mse', optimizer = 'sgd') # 指定 loss 值的计算方法和优化器
33 model_3.compile(loss = tf.losses.mean_squared_error, optimizer = 'sgd')

```

这里搭建模型 `model` 与 `model_3` 的反向传播的网络。在实现的过程中，指定的损失函数与优化器既可以用字符串形式的传入（见代码第 32 行），也可以用 TensorFlow 中的函数形式传入（见代码第 33 行）。

在代码第 32 行用到的字符串可以在 Keras 的帮助文档（见 6.1.6 小节的帮助文档链接）中找到。

## 6.7.4 代码实现：用两种方法训练模型

训练模型可以使用集成度较低的 `train_on_batch` 方法，也可以使用集成度较高的 `fit` 方法。具体代码如下：

代码 6-11 keras 回归模型（续）

```
34 for step in range(20):
35     cost = model.train_on_batch(train_data[0], train_data[1]) #训练模型，返回损失值
36     if step % 10 == 0:
37         print ('loss: ', cost)
38
39 #直接使用 fit 函数来训练
40 model_3.fit(x=train_data[0],y=train_data[1], batch_size=10, epochs=20)
```

代码第 34~37 行，用 `for` 循环训练模型。每调用一次 `train_on_batch`，优化器便反向训练一次。

代码第 40 行，直接用 `fit` 方法进行训练。在指定好迭代的次数和批次后，会自动完成循环迭代。

代码运行后，输出以下结果：

```
loss: 1.0861262
.....
loss: 0.1734276
Epoch 1/20
100/100 [=====] - 0s 5ms/step - loss: 1.8135
.....
Epoch 20/20
100/100 [=====] - 0s 191us/step - loss: 0.3026
```

在输出结果中，前 3 行是使用 `train_on_batch` 方法训练模型的输出，后面几行是调用 `fit` 方法的输出。

## 6.7.5 代码实现：获取模型参数

对于训练好的模型，可以用 `get_weights` 方法获取参数。直接用 `Model` 类创建的网络与用 `Sequential` 类创建的网络，两者在使用 `get_weights` 方法时会有所不同。下面通过代码演示。

代码 6-11 keras 回归模型（续）

```
41 W,b= model.get_weights() #直接使用 Model 类定义模型
42 print ('Weights: ',W)
43 print ('Biases: ', b)
44 #指定具体层来获取参数
45 W, b = model_3.layers[0].get_weights()
46 print ('Weights: ',W)
47 print ('Biases: ', b)
```

比较代码第41与第45行可以看出：用 Sequential 类创建的网络，还可以指定提取某一层的权重；直接用 Model 类创建网络，只能将全部权重一次全部提取出来。

代码运行后，输出以下结果：

```
Weights: [[1.4668063]]
Biases: [-0.01882044]
Weights: [[1.071188]]
Biases: [-0.00182833]
```

在输出结果中，前两行是直接用 model 类定义的网络权重，后两行是用 Sequential 类定义模型的权重。

## 6.7.6 代码实现：测试模型与用模型进行预测

与估算器的方法类似，tf.keras 接口中也有 evaluate 方法与 predict 方法。前者用于测试，后者用于预测。

下面通过代码演示这两种方法的使用。

### 代码 6-11 keras 回归模型（续）

---

```
48 cost = model.evaluate(train_data[0], train_data[1], batch_size = 10) # 测试
49 print ('test loss: ', cost)
50
51 a = model.predict(train_data[0], batch_size = 10) # 预测
52 print(a[:10])
53 print(train_data[1][:10])
```

---

代码运行后，输出以下结果：

```
100/100 [=====] - 0s 3ms/step test loss:
0.1835745729506016
[[[-1.4856267]
...
[-1.2189347]
[-2.03062256 .....-1.6202334 ]
```

第1行是测试的输出结果，后面几行是预测的输出结果。

## 6.7.7 代码实现：保存模型与加载模型

tf.keras 接口保留了与 Keras 框架中保存模型的格式，可以生成扩展名为“.h5”的模型文件，也可以生成 TensorFlow 框架中检查点格式的模型文件。

### 1. 生成及载入 h5 模型文件

模型训练好之后，可以用 save 方法进行保存。保存后的模型文件可以通过函数 load\_model 进行载入。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```

54 model.save('my_model.h5')          #保存模型
55 del model                         #删除当前模型
56 model = tf.keras.models.load_model('my_model.h5') #加载模型
57 a = model.predict(train_data[0], batch_size = 10)
58 print("加载后的测试",a[:10])

```

上面代码演示了一个保存模型并二次加载进行使用的过程。代码运行后，输出以下结果：

```

加载后的测试 [[-1.4856267]
.....
[-1.2189347]]

```

可以看到模型能够正常预测，这表示其已经被成功加载了。在本地代码的同级目录下，生成了模型文件“my\_model.h5”。

**提示：**

h5 文件属于 h5py 类型，可以直接手动调用 h5py 进行解析。例如，下列代码可以将模型中的节点显示出来：

```

import h5py
f=h5py.File('my_model.h5')
for name in f:
    print(name)

```

运行后，会输出以下结果：

model_weights	#模型的权重
optimizer_weights	#优化器的权重

**2. 生成 TensorFlow 格式的模型文件**

调用 save\_weights 方法，可以生成 TensorFlow 检查点格式的文件。在 save\_weights 方法中，可以根据 save\_format 参数对应的格式生成指定的模型文件。

参数 save\_format 的取值有两种：“tf”与“h5”。前者是 TensorFlow 检查点格式，后者是 Keras 检查点格式。

在不指定参数 save\_format 的情况下，如果 save\_weights 中的文件名不是以“.h5”或“.keras”结尾，则会生成 TensorFlow 检查点格式的文件，否则会生成 Keras 框架格式的模型文件。具体代码如下：

**代码 6-11 keras 回归模型（续）**

```

59 #生成tf格式的模型
60 model.save_weights('./keraslog/kerasmodel') #默认生成tf格式的模型
61 #生成tf格式的模型，手动指定
62 os.makedirs("./kerash5log", exist_ok=True)
63 model.save_weights('./kerash5log/kerash5model', save_format = 'h5')#可以指定
      save_format 是h5 或tf 来生成对应的格式

```

代码运行后，系统会在本地的 keraslog 文件夹下生成 TensorFlow 检查点格式的文件，在本地的 kerash5log 文件夹下生成 Keras 框架格式的模型文件 kerash5model（虽然没有扩展名，但它是 h5 格式）。



#### 提示：

将 Keras 框架格式的模型文件转化成 TensorFlow 检查点的模型文件，这个过程是单向的。目前 TensorFlow 的版本中，还没有提供将 TensorFlow 检查点格式的文件转化成 Keras 格式的模型文件的方法。

### 6.7.8 代码实现：将模型导出成 JSON 文件，再将 JSON 文件导入模型

TensorFlow 的检查点文件中包含模型的符号及对应的值。而 Keras 框架中生成的检查点文件（扩展名为 h5 的文件）只包含模型的值。

在 tf.keras 接口中，可以将模型符号转化为 JSON 文件再进行保存。具体代码如下：

代码 6-11 keras 回归模型（续）

```
64 json_string = model.to_json() #模型 JSON 化，等价于 json_string =
65   model.get_config()
66 open('my_model.json', 'w').write(json_string)
67 #加载模型数据和 weights
68 model_7 = tf.keras.models.model_from_json(open('my_model.json').read())
69 model_7.load_weights('my_model.h5')
70 a = model_7.predict(train_data[0], batch_size = 10)
71 print("加载后的测试",a[:10])
```

上述代码实现的逻辑如下：

- (1) 将模型符号保存到 my\_model.json 文件中。
- (2) 从 my\_model.json 文件中载入权重到模型 model\_7 中。
- (3) 为模型 model\_7 恢复权重。
- (4) 用模型 model\_7 进行预测。

代码运行后，输出以下结果：

```
加载后的测试 [[-1.4856267]
...
[-1.2189347]]
```

可以看到，程序成功载入模型的符号及权重，并能够执行预测任务。



#### 提示：

用 tf.keras 接口开发模型时，常会把模型文件分成 JSON 和 h5 两种格式存储，用于不同的场景：

- 在使用场景中，直接载入 h5 模型文件。
- 在训练场景中，同时载入 JSON 与 h5 两个模型文件。

这种做法可以让模型训练场景与使用场景分离。通过隐藏源码的方式保证代码版本的唯一性（防止使用者修改模型而产生多套模型源码，难以维护），是合作项目中很常见的技巧。

### 6.7.9 实例 23：在 tf.keras 接口中使用预训练模型 ResNet

在 tf.keras 接口中也预制了许多训练好的成熟模型，其中包括了在 imagenet 数据集上训练好的 densenet、NASNet、mobilenet 等扩展名为 h5 的模型。

#### 1. 获取预训练模型

具体地址如下：

```
https://github.com/fchollet/deep-learning-models/releases
```

每一种模型会有两个文件：一个是正常模型文件，另一个是以 no-top 结尾的文件。例如，resnet50 文件如下：

```
resnet50_weights_tf_dim_ordering_tf_kernels.h5  
resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
```

其中，以“no-top”结尾的文件是提取特征的模型，用于微调模型或嵌入模型的场景；而正常的模型文件（NASNet-large.h5）直接用于预测场景。

在下载预训练模型时，如果 Keras 框架的后端运行在 Theano 框架（另一种支持 Keras 前端的深度学习框架）上，则需要将文件名称中间的“tf”换成“th”。例如：

```
resnet50_weights_th_dim_ordering_th_kernels.h5  
resnet50_weights_th_dim_ordering_th_kernels_notop.h5
```

在 Theano 框架上运行的 Keras 模型文件，与在 TensorFlow 框架上运行的 Keras 模型文件最大的区别是：图片维度的默认顺序不同。在 Theano 框架中，图片的通道维度在前，例如(3,224,224)；而在 TensorFlow 中，图片通道维度在后，例如(224,224,3)。

#### 2. 使用预训练模型

下面通过预训练模型 ResNet 来识别图片。

用 tf.keras 接口可以非常方便地预测模型，只需要几行代码。

#### 代码 6-12 用 tf.keras 预训练模型

```
01 from tensorflow.python.keras.applications.resnet50 import ResNet50  
02 from tensorflow.python.keras.preprocessing import image  
03 from tensorflow.python.keras.applications.resnet50 import preprocess_input,  
    decode_predictions  
04 import numpy as np  
05  
06 model = ResNet50(weights='imagenet') # 创建 ResNet 模型
```

```

07 #载入图片进行处理
08 img_path = 'hy.jpg'
09 img = image.load_img(img_path, target_size=(224, 224))
10 x = image.img_to_array(img)
11 x = np.expand_dims(x, axis=0)
12 x = preprocess_input(x)
13
14 preds = model.predict(x) #使用模型预测
15 print('Predicted:', decode_predictions(preds, top=3)[0]) #输出结果

```

执行第6行代码时，会从网上下载模型文件并载入。

执行第14行代码时，会从网上下载类名文件并载入。

整个代码运行后，输出以下结果：

```

.....
Downloading data from https://s3.amazonaws.com/deep-learning-models/image-models/
imagenet_class_index.json
40960/35363 [=====] - 2s 37us/step
Predicted: [('n03642806', 'laptop', 0.46727782), ('n03617480', 'kimono', 0.04840326),
('n03782006', 'monitor', 0.04691172)]

```

在结果中，前4行是下载类名文件，最后两行是显示结果。

该例子中使用的图片与第3章的一致，见图3-5(a)。预测结果为laptop(笔记本电脑)。



### 提示：

改代码可以直接在TensorFlow 1.x版本和TensorFlow 2.x版本中运行。

### 3. 手动下载预训练模型

如果由于网络原因导致下载模型较慢，则可以手动下载，地址如下：

[https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50\\_weights\\_tf\\_dim\\_ordering\\_tf\\_kernels.h5](https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels.h5)

将加载好的模型放到本地，将第6行代码改成以下即可：

#### 代码6-12 用tf.keras预训练模型（片段）

```

06 model = ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels.
h5')

```

该代码的作用是，让ResNet50模型从指定的模型文件加载权重。

如果使用的是自己的模型，则可以按照以下参数来构建模型：

```

def ResNet50(include_top=True, #是否返回顶层结果。False代表返回特征
            weights='imagenet', #加载权重路径
            input_tensor=None, #输入张量，用于嵌入的其他网络中
            input_shape=None, #输入的形状
            pooling=None, #可以取值avg、max，对返回特征进行（全局平均、最大）池化操作
            classes=1000): #分类个数

```

### 6.7.10 扩展：在动态图中使用 tf.keras 接口

在 tf.keras 接口中，训练和使用模型的方法与在估算器中的方法很类似，即对模型使用流程的高度集成化封装。所以这种方式无法适用于精细化调节模型的场景。

将 6.7.9 小节中的代码稍做改变，即可将其改为动态图框架中的代码。具体做法如下：

- (1) 在 6.7.9 小节的代码第 5 行，添加动态图启动函数 `tf.enable_eager_execution()`。
- (2) 修改 6.7.9 小节的代码第 14 行，将 `tf.keras` 模型的 `predict` 方法改成直接在动态图里使用模型的方式。
- (3) 修改 6.7.9 小节的代码第 15 行，将结果 `preds` 打印出来。

具体代码如下：

**代码 6-12 用 tf.keras 预训练模型（片段）**

```
05 tf.enable_eager_execution()
.....
14 preds = model(x)
15 print('Predicted:', decode_predictions(preds.numpy(), top=3)[0])
```

如果是在 TensorFlow 2.x 版本中运行，则还需要将代码第 5 行删掉，不需要再额外执行启动动态图的代码。

### 6.7.11 实例 24：在静态图中使用 tf.keras 接口

本实例将 6.7.9 小节的用法改写成在静态图中调用 `tf.keras` 接口的方式。

具体代码如下：

**代码 6-13 在静态图中使用 tf.keras**

```
01 import tensorflow as tf
02 import matplotlib.pyplot as plt
03 from tensorflow.python.keras.applications.resnet50 import ResNet50
04 from tensorflow.python.keras.preprocessing import image
05 from tensorflow.python.keras.applications.resnet50 import preprocess_input,
   decode_predictions
06
07 inputs = tf.placeholder(tf.float32, (224, 224, 3)) # 定义占位符
08
09 tensorimg = tf.expand_dims(inputs, 0) # 预处理
10 tensorimg = preprocess_input(tensorimg)
11
12 with tf.Session() as sess: # 在会话 (session) 中运行
13     sess.run(tf.global_variables_initializer())
14
15     Reslayer =
16         ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels.h5') # 模型
17
18     logits = Reslayer(tensorimg)
```

```

17
18     img_path = 'dog.jpg'
19     img = image.load_img(img_path, target_size=(224, 224))
20     logitsv = sess.run(logits, feed_dict={inputs: img})
21     Pred = decode_predictions(logitsv, top=3)[0]
22     print('Predicted:', Pred, len(logitsv[0]))
23
24 #可视化
25 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 8))
26 fig.sca(ax1)
27 ax1.imshow(img)
28 fig.sca(ax2)
29
30 barlist = ax2.bar(range(3), [i[2] for i in Pred])
31 barlist[0].set_color('g')
32
33 plt.sca(ax2)
34 plt.ylim([0, 1.1])
35 plt.xticks(range(3), [i[1][:15] for i in Pred], rotation='vertical')
36 fig.subplots_adjust(bottom=0.2)
37 plt.show()

```

直接将 ResNet50 模型当成运行图中的一层即可（见代码第 21 行），这样由 ResNet50 模型构成的网络节点同样可以用占位符和会话形式运行。代码运行后，输出如下结果：

```
Predicted: [('n02109961', 'Eskimo_dog', 0.5246922), ('n02110185', 'Siberian_husky', 0.47256017), ('n02091467', 'Norwegian_elkhound', 0.0011198776)] 1000
```

可视化的结果如图 6-4 所示。

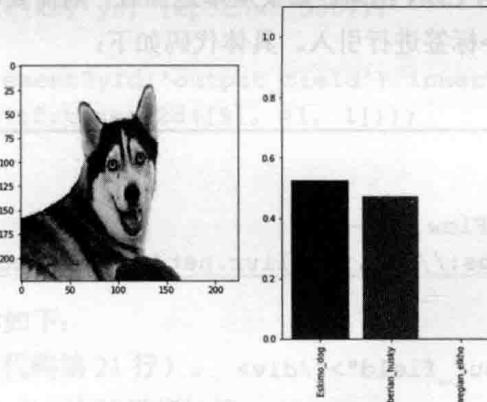


图 6-4 浏览器中回归模型返回的结果



**提示：**

因为 ResNet50 模型需要加载预训练模型（见代码第 21 行），所以加载模型的过程必须放到初始化过程（见代码第 19 行）之后。如果放到初始化过程之前，则在初始化时会将加载

的权重清掉，从而导致模型无法正常输出预测结果。

另外，在使用 ResNet50 模型时，也可以将输入指定成占位符的形式。例如代码第 21、22 行，如果用下列代码替换，会得到一样的效果。

```
Reslayer = ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels.h5',
                     input_tensor=tensorimg,input_shape = (224, 224, 3)) #指定输入层
logits = Reslayer.layers[-1].output #指定输出层
print(logits)
```

## 6.8 实例 25：用 tf.js 接口后方训练一个回归模型

本实例是一个最简单的 tf.js 接口使用实例，来演示如何用 JS 脚本训练模型。

### 实例描述

在浏览器中，调用 TensorFlow 的 API，从而在两组看似混乱的数据中学习规律并进行拟合，找到其中的对应关系，并通过输入任意值进行预测。

该例子来源与 tf.js 接口的示例教程，具体链接如下：

<https://github.com/tensorflow/tfjs>

### 6.8.1 代码实现：在 HTTP 的头标签中添加 tfjs 模块

首先创建一个空的 txt 文件，并将其改名为“6-14\_tfjs 回归例子.html”，然后在文件中添加引入 JS 的头文件。

这里使用的 tfjs 文件来自 CDN 网络。如果从本地加载，则需要将其放到本地站点对应的路径下。JS 脚本要通过<script>标签进行引入。具体代码如下：

#### 代码 6-14 tfjs 回归例子

```
01 <html>
02   <head>
03     <!-- Load TensorFlow.js -->
04     <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>
05   </head>
06   <body>
07     <div id="output_field"></div>
08   </body>
```

上面的代码有两部分：一部分是<head>，另一部分是<body>。在<body>中定义了一个 div，用于输出最终的结果。

## 6.8.2 代码实现：用JavaScript脚本实现回归模型

HTML中的JS是通过<script>标签来标记的。在<script>中添加了一个函数learnLinear，实现模型的训练与预测。

在JavaScript中建立网络模型的语法，与Keral的语法几乎相同：都是通过一个model对象实现基本结构，并通过模型model的fit方法进行训练，通过模型model的predict方法进行使用。

在learnLinear函数中完成了以下步骤：

- (1) 建立一个全连接网络。
- (2) 以平方差的方式计算损失值。
- (3) 设置优化器为sgd。
- (4) 手动输入模拟数值作为样本（这里模拟样本x、y值的规律为 $y=2x-1$ ）。
- (5) 完成模型的训练。

具体代码如下：

代码 6-14 tfjs 回归例子（续）

```

09 <script>
10   async function learnLinear(){
11     const model = tf.sequential();
12     model.add(tf.layers.dense({units: 1, inputShape: [1]}));
13     model.compile( {loss: 'meanSquaredError', optimizer: 'sgd'} );
14
15     const xs = tf.tensor2d([-1, 0, 1, 2, 3, 4], [6, 1]);
16     const ys = tf.tensor2d([-3, -1, 1, 3, 5, 7], [6, 1]);
17
18     await model.fit(xs, ys, {epochs: 500});
19
20     document.getElementById('output_field').innerText =
21       model.predict(tf.tensor2d([9], [1, 1]));
22   }
23   learnLinear();
24 </script>
25 <html>
```

代码第20~23行的解读如下：

- (1) 将9传入模型（见代码第21行）。
- (2) 用模型model的predict方法进行计算。
- (3) 将模型的输出结果放入网页的div节点“output\_field”中。
- (4) 用learnLinear函数使其运行（见代码第23行）。

## 6.8.3 运行程序：在浏览器中查看效果

在Windows操作系统中双击网页文件“6-14\_tfjs 回归例子.html”，系统会自动用浏览器

打开该网页文件，如图 6-5 所示。

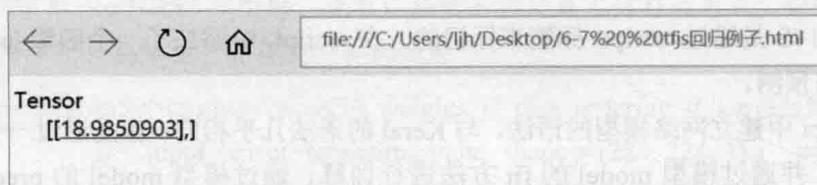


图 6-5 浏览器中的回归模型返回结果

tf.js 接口能让 TensorFlow 编写的 AI 模型以 Web 应用程序的方式运行在浏览器终端。这进一步提升了部署的灵活性。

本书的重点是基于 Python 语言进行开发。这里只介绍一个最简单的实例。用好 tf.js 接口还需要有扎实的 JavaScript 编程知识才可以。

#### 6.8.4 扩展：tf.js 接口的应用场景

用 tf.js 接口开发的程序是在浏览器中运行的。模型使用了客户端的运算资源。在部署应用程序时，这种方案可以分担后端服务器的运算压力。

但是，用 tf.js 接口开发的应用程序在浏览器中运行时，浏览器内部会将模型下载到本地。如果模型文件太大，则会严重影响用户体验。

总结：用 tf.js 接口开发的应用程序适用于模型文件比较小、并发量很大的场景。在实际使用中，如果模型较大，则可以将模型拆成前处理和后处理两部分，并将前处理部分放到 tf.js 接口中去运行，让用户终端来分担一些后端的运算压力。

### 6.9 实例 26：用估算器框架实现分布式部署训练

本实例使用与 6.4 节一样的数据与模型进行分布式演示。

#### 实例描述

假设有这么一组数据集，其  $x$  和  $y$  的对应关系是  $y \approx 2x$ 。

训练模型来学习这些数据集，使模型能够找到其中的规律，即让神经网络自己能够总结出  $y \approx 2x$  这样的公式。

要求用估算器框架来实现，并完成分布式部署训练。

#### 6.9.1 运行程序：修改估算器模型，使其支持分布式

在 6.4 节中，将 6.4.8 小节以前的代码全部复制过来，并在后面用 `tf.estimator.train_and_evaluate` 方法分布式训练模型。

具体代码如下：

### 代码 6-15 用估算器框架进行分布式训练

```
.....  
27 estimator =  
    tf.estimator.Estimator(model_fn=my_model, model_dir='myestimatormode', pa  
    rams={'learning_rate': 0.1},  
    config=tf.estimator.RunConfig(session_config=session_config))  
28  
29 #创建TrainSpec与EvalSpec  
30 train_spec = tf.estimator.TrainSpec(input_fn=lambda:  
    train_input_fn(train_data, batch_size), max_steps=1000)  
31 eval_spec = tf.estimator.EvalSpec(input_fn=lambda:  
    eval_input_fn(test_data, None, batch_size))  
32  
33 tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

## 6.9.2 通过TF\_CONFIG进行分布式配置

通过添加TF\_CONFIG变量实现分布式训练的角色配置。添加TF\_CONFIG变量有两种方法。

- 方法一：直接将TF\_CONFIG添加到环境变量里。
- 方法二：在程序运行前加入TF\_CONFIG的定义。例如在命令行里输入：

```
TF_CONFIG='内容' python xxxx.py
```

在上面的两种方法任选其一即可。在添加完TF\_CONFIG变量之后，还要为其指定内容。具体格式如下。

### 1. TF\_CONFIG 内容格式

变量TF\_CONFIG的内容是一个字符串。该字符串用于描述分布式训练中各个角色(chief、worker、ps)的信息。每个角色都由task里面的type来指定。具体代码如下。

(1) chief角色：分布式训练的主计算节点。

```
TF_CONFIG='{'  
    "cluster": {  
        "chief": ["主机0-IP: 端口"],  
        "worker": ["主机1-IP: 端口", "主机2-IP: 端口", "主机3-IP: 端口"],  
        "ps": ["主机4-IP: 端口", "主机5-IP: 端口"]  
    },  
    "task": {"type": "chief", "index": 0}  
'}
```

(2) worker角色：分布式训练的一般计算节点。

```
TF_CONFIG='{'  
    "cluster": {  
        "chief": ["主机0-IP: 端口"],  
        "worker": ["主机1-IP: 端口", "主机2-IP: 端口", "主机3-IP: 端口"],  
    },  

```

```

    "ps": ["主机 4-IP: 端口", "主机 5-IP: 端口"],
},
"task": {"type": "worker", "index": 0}
}

```

(3) ps 角色：分布式训练的服务端。

```

TF_CONFIG='{
  "cluster": {
    "chief": ["主机 0-IP: 端口"],
    "worker": ["主机 1-IP: 端口", "主机 2-IP: 端口", "主机 3-IP: 端口"],
    "ps": ["主机 4-IP: 端口", "主机 5-IP: 端口"]
  },
  "task": {"type": "ps", "index": 0}
}'

```

有关这部分的更多内容，还可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 4.5 节。

## 2. 代码实现：定义 TF\_CONFIG 的环境变量

本实例只是一个演示程序，将三种角色放在了同一台机器上运行。具体步骤如下：

(1) 将 TF\_CONFIG 的环境变量放到代码里。

(2) 将代码文件复制成 3 份，分别代表 chief、worker、ps 三种角色。

其中，代表 ps 角色的具体代码如下：

**代码 6-16 用估算器框架分布式训练 ps**

```

01 TF_CONFIG=''{{
02   "cluster": {
03     "chief": ["127.0.0.1:2221"],
04     "worker": ["127.0.0.1:2222"],
05     "ps": ["127.0.0.1:2223"]
06   },
07   "task": {"type": "ps", "index": 0}
08 }''

09
10 import os
11 os.environ['TF_CONFIG']=TF_CONFIG
12 print(os.environ.get('TF_CONFIG'))
.....
```

该代码是 ps 角色的主要实现。将第 7 行中的 ps 改为 chief，得到代码文件“6-17 用估算器框架进行分布式训练 chief.py”，用于创建 chief 角色。具体代码如下：

```
"task": {"type": "chief", "index": 0}
```

再将第 7 行中的 ps 改为 worker，得到代码文件“6-18 用估算器框架进行分布式训练 worker.py”，用于创建 worker 角色。具体代码如下：

```
"task": {"type": "worker", "index": 0}
```

### 6.9.3 运行程序

在运行程序之前，需要打开3个Console（控制台），如图6-6所示。第1个是ps角色，第2个是chief角色，第3个是worker角色。

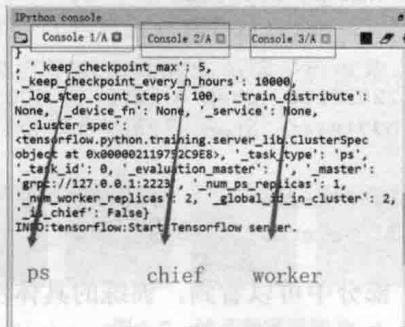


图6-6 打开3个控制台

按照图6-6中控制台的具体顺序，依次运行每个角色的代码文件。生成的结果如下：

(1) 控制台 Console1：用于展示ps角色。启动后等待chief与worker的接入。

```
.....
'_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at
0x000002119752C9E8>, '_task_type': 'ps', '_task_id': 0, '_evaluation_master': '',
'_master': 'grpc://127.0.0.1:2223', '_num_ps_replicas': 1, '_num_worker_replicas': 2,
'_global_id_in_cluster': 2, '_is_chief': False}
INFO:tensorflow:Start Tensorflow server.
```

(2) 控制台 Console2：用于展示chief角色。在训练完成后保存模型。

```
.....
'_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at
0x0000025AD5B8B9E8>, '_task_type': 'chief', '_task_id': 0, '_evaluation_master': '',
'_master': 'grpc://127.0.0.1:2221', '_num_ps_replicas': 1, '_num_worker_replicas': 2,
'_global_id_in_cluster': 0, '_is_chief': True}
.....
INFO:tensorflow:loss = 0.13062291, step = 2748 (0.367 sec)
INFO:tensorflow:global_step/sec: 565.905
INFO:tensorflow:global_step/sec: 532.612
INFO:tensorflow:loss = 0.11379747, step = 2953 (0.372 sec)
INFO:tensorflow:global_step/sec: 578.003
INFO:tensorflow:global_step/sec: 578.006
INFO:tensorflow:loss = 0.11819798, step = 3157 (0.353 sec)
INFO:tensorflow:global_step/sec: 574.74
INFO:tensorflow:global_step/sec: 558.949
.....
INFO:tensorflow:loss = 0.09850123, step = 5814 (0.424 sec)
INFO:tensorflow:global_step/sec: 572.337
INFO:tensorflow:global_step/sec: 439.875
INFO:tensorflow:Saving checkpoints for 6002 into myestimatormode\model.ckpt.
INFO:tensorflow:Loss for final step: 0.04346009.
```

(3) 控制台 Console3：用于展示 worker 角色。只负责训练。

```
.....
<tensorflow.python.training.server_lib.ClusterSpec object at 0x00000209A423D9E8>,
'_task_type': 'worker', '_task_id': 0, '_evaluation_master': '', '_master':
'grpc://127.0.0.1:2222', '_num_ps_replicas': 1, '_num_worker_replicas': 2,
'_global_id_in_cluster': 1, '_is_chief': False}
.....
INFO:tensorflow:loss = 0.22635186, step = 2292 (0.408 sec)
INFO:tensorflow:loss = 0.07718446, step = 2457 (0.329 sec)
.....
INFO:tensorflow:loss = 0.1483176, step = 5982 (0.405 sec)
INFO:tensorflow:Loss for final step: 0.08431114.
```

从输出结果的(2)、(3)部分中可以看到，训练的具体步数(step)并不是连续的，而是交叉进行的。这表示，chief 角色与 worker 角色二者在一起进行了协同训练。

#### 6.9.4 扩展：用分布策略或 KubeFlow 框架进行分布式部署

在实际场景中，还可以用分布策略或 KubeFlow 框架进行分布式部署。其中，分布策略的方法介绍可以参考 6.1.9 小节，KubeFlow 框架的使用方法可以参考以下链接：

<https://www.kubeflow.org/>

### 6.10 实例 27：在分布式估算器框架中用 tf.keras 接口训练 ResNet 模型，识别图片中是橘子还是苹果

在估算器框架中使用 train\_and\_evaluate 方法是一个非常便捷的开发方案。可以根据实际情况自由部署：

- 如果训练量小，则可以直接在本机上运行。
- 如果训练量大，则可以通过添加环境变量的方式在多台机器上分布训练。

本实例就用 train\_and\_evaluate 方法对预训练模型进行微调。

#### 实例描述

有一组包含苹果和橘子的图片数据集。通过微调预训练模型，使模型能够识别出图片中是苹果还是橘子。

在样本量不足的情况下，最快捷的方式就是对预训练模型进行微调。在 6.7.9 小节介绍过，tf.keras 接口中可以有好多预训练好的模型供微调使用。这里以 ResNet50 模型为例，演示其具体的用法。

#### 6.10.1 样本准备

该实例的样本是各种各样的橘子和苹果的图片。样本下载地址如下：

[https://people.eecs.berkeley.edu/~taesung\\_park/CycleGAN/datasets/](https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/)

将样本下载后，放到本地代码的同级目录下即可。该样本结构与4.7节实例中的样本结构几乎一样。

在样本处理环节，可以直接重用4.7节数据集部分的代码：

- (1) 将4.7节数据集部分的代码复制到本地。
- (2) 修改数据集路径，使其指向本地的苹果橘子数据集。

运行程序后可以看到输出的结果，如图6-7所示。

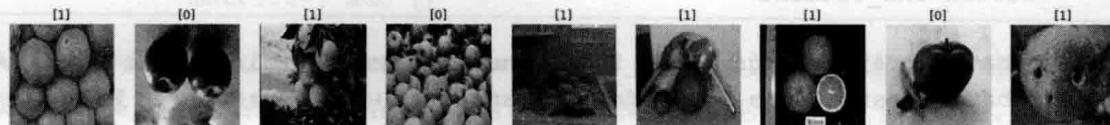


图6-7 橘子和苹果样本

### 6.10.2 代码实现：准备训练与测试数据集

将4.7节的实例中的代码文件“4-10 将图片文件制作成Dataset数据集.py”复制到本地代码的同级目录下，修改其中的图片归一化函数\_norm\_image，具体代码如下：

#### 代码6-19 用ResNet识别橘子和苹果

```
01 def _norm_image(image, size, ch=1, flattenflag = False): # 定义函数，实现数据归
    一化处理
02     image_decoded = image/127.5-1
03     if flattenflag==True:
04         image_decoded = tf.reshape(image_decoded, [size[0]*size[1]*ch])
05     return image_decoded
```

### 6.10.3 代码实现：制作模型输入函数

制作模型的输入函数，并对其进行测试。具体代码如下：

#### 代码6-19 用ResNet识别橘子和苹果（续）

```
06 from tensorflow.python.keras.preprocessing import image
07 from tensorflow.python.keras.applications.resnet50 import ResNet50
08 from tensorflow.python.keras.applications.resnet50 import preprocess_input,
    decode_predictions
09
10 size = [224,224] # 图片尺寸
11 batchsize = 10 # 批次大小
12
13 sample_dir=r"./apple2orange/train"
14 testsample_dir = r"./apple2orange/test"
15
16 traindataset = dataset(sample_dir,size,batchsize) # 训练集
```

```

17 testdataset = dataset(testsample_dir, size, batchsize, shuffleflag = False) # 测试集
18
19 print(traindataset.output_types) # 打印数据集的输出信息
20 print(traindataset.output_shapes)
21
22 def imgs_input_fn(dataset):
23     iterator = dataset.make_one_shot_iterator() # 生成一个迭代器
24     one_element = iterator.get_next() # 从 iterator 里取一个元素
25     return one_element
26
27 next_batch_train = imgs_input_fn(traindataset) # 从 traindataset 里取一个元素
28 next_batch_test = imgs_input_fn(testdataset) # 从 testdataset 里取一个元素
29     if flattenflag==True:
30 with tf.Session() as sess: # 建立会话 (session)
31     sess.run(tf.global_variables_initializer()) # 初始化
32     try:
33         for step in np.arange(1):
34             value = sess.run(next_batch_train)
35             showimg(step,value[1],np.asarray(
36                 (value[0]+1)*127.5,np.uint8),10) # 显示图片
37     except tf.errors.OutOfRangeError: # 捕获异常
38         print("Done!!!")

```

代码第 30 行是用会话 (session) 对输入函数进行测试。运行后，如果看到如图 6-7 所示的效果，则表示输入函数正确。

#### 6.10.4 代码实现：搭建 ResNet 模型

搭建 ResNet 模型的步骤如下：

- (1) 手动将预训练模型文件“resnet50\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5”下载到本地（也可以采用 6.7.9 小节的方法——在程序执行时通过设置让其自动从网上下载）。
- (2) 用 tf.keras 接口加载 ResNet50 模型，并将其作为一个网络层。
- (3) 用 tf.keras.models 类在 ResNet50 层之后添加两个全连接网络层。
- (4) 用激活函数 sigmoid 对模型最后一层的结果进行处理，得出最终的分类结果：是桔子还是苹果。

具体代码如下：

#### 代码 6-19 用 ResNet 识别橘子和苹果（续）

```

39 img_size = (224, 224, 3)
40 inputs = tf.keras.Input(shape=img_size)
41 conv_base =
    ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5',
    input_tensor=inputs, input_shape = img_size , include_top=False) # 创建 ResNet
42

```

```

43 model = tf.keras.models.Sequential()          # 创建整个模型
44 model.add(conv_base)
45 model.add(tf.keras.layers.Flatten())
46 model.add(tf.keras.layers.Dense(256, activation='relu'))
47 model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
48 conv_base.trainable = False                  # 不训练ResNet的权重
49 model.summary()
50 model.compile(loss='binary_crossentropy',   # 构建反向传播
51             optimizer=tf.keras.optimizers.RMSprop(lr=2e-5),
52             metrics=['acc'])

```

代码第48行，通过将ResNet50层(conv\_base)的权重设为不可训练，固定ResNet50层的权重，让其只输出图片的特征结果，并用该特征结果去训练后面的两个全连接层。

## 6.10.5 代码实现：训练分类器模型

训练分类器模型的步骤如下：

- (1) 用tf.keras.estimator.model\_to\_estimator方法创建估算器模型est\_app2org。
- (2) 用train\_and\_evaluate方法对估算器模型est\_app2org进行训练。

具体代码如下：

### 代码 6-19 用ResNet识别橘子和苹果（续）

```

53 model_dir = "./models/app2org"
54 os.makedirs(model_dir, exist_ok=True)
55 print("model_dir: ", model_dir)
56 est_app2org = tf.keras.estimator.model_to_estimator(keras_model=model,
57   model_dir=model_dir)
58 # 训练模型
59 train_spec = tf.estimator.TrainSpec(input_fn=lambda:
60   imgs_input_fn(traindataset),
61   max_steps=500)
62 eval_spec = tf.estimator.EvalSpec(input_fn=lambda:
63   imgs_input_fn(testdataset))
64
65 import time
66 start_time = time.time()
67 tf.estimator.train_and_evaluate(est_app2org, train_spec, eval_spec)
68 print("--- %s seconds ---" % (time.time() - start_time))

```

代码第60行，指定了迭代训练的次数是500次。还可以通过增大训练次数的方式提高模型的精度。如果想要缩短训练时间，则可以运用6.9节的知识在多台机器上进行分布训练。

代码运行后，在本地路径“models\app2org”下生成了检查点文件。该文件是最终的结果。

### 6.10.6 运行程序：评估模型

评估模型的代码实现部分与 6.4 节几乎一样，只是需要将 `estimator.train` 方法替换成 `tf.estimator.train_and_evaluate` 方法。

具体代码如下：

代码 6-19 用 ResNet 识别橘子和苹果（续）

```

67 img = value[0] #准备评估数据
68 lab = value[1]
69
70 pre_input_fn =
    tf.estimator.inputs.numpy_input_fn(img,batch_size=10,shuffle=False)
71 predict_results = est_app2org.predict( input_fn=pre_input_fn) #评估输入的图片
72
73 predict_logits = [] #处理评估结果
74 for prediction in predict_results:
75     print(prediction)
76     predict_logits.append(prediction['dense_1'][0])
77 #可视化结果
78 predict_is_org = [int(np.round(logit)) for logit in predict_logits]
79 actual_is_org = [int(np.round(label[0])) for label in lab]
80 showimg(step,value[1],np.asarray((value[0]+1)*127.5,np.uint8),10)
81 print("Predict :",predict_is_org)
82 print("Actual  :",actual_is_org)

```

代码第 67、68 行将数组 `value` 分成图片和标签，作为待输入的样本数据。数组 `value` 是通过代码第 34 行从输入函数中取出的。

在实际应用中，第 67、68 行的代码还需要被换成真正的待测数据。代码运行后，可以看到评估结果，如图 6-8 所示。



图 6-8 模型的评估结果

输出的预测结果与真实值如下：

```

Predict: [0, 1, 1, 1, 0, 1, 1, 1, 1, 0]
Actual: [0, 1, 1, 1, 0, 1, 1, 1, 1, 1]

```

### 6.10.7 扩展：全连接网络的优化

如要想获得更高的精度，则除增加训练次数外，还可以使用以下优化方案：

- 在模型最后两层全连接网络中，加入 `dropout` 方法和正则化方法，使模型具有更好的泛

化能力。

- 将模型最后两层全连接的网络结构改成“一层全尺度卷积与一层 $1\times 1$ 卷积组合”的结构（见8.7.19小节“1. 实现分类器”的代码实现部分）。
- 在数据集处理部分，对图片做更多的增强变换。

有兴趣的读者可以自行尝试。

## 6.11 实例28：在T2T框架中用tf.layers接口实现MNIST数据集分类

T2T是google基于TensorFlow新开源的深度学习库。该库将深度学习所需要的元素（数据集、模型、学习率、超参数等）封装成标准化的统一接口，使用起来更加方便。

### 实例描述

有一个MNIST数据集，其中包含0~9之间的手写数字图片。要求在T2T框架中用tf.layers接口将这些数字识别出来。

MNIST数据集属于深度学习领域使用最广的测试数据集。该例子用一个简单的卷积模型在MNIST上完成分类任务。在此过程中，重点演示如何在T2T框架中用统一的数据集、模型等接口进行训练。

### 6.11.1 代码实现：查看T2T框架中的数据集（problems）

在T2T框架中，将数据集统一命名为problems。一个problems代表一个具体的数据集。

在按照6.1.14小节的方式安装好T2T框架之后，可以通过以下代码在T2T框架中查找其内部集成好的数据集。

#### 代码6-20 在T2T框架中训练mnist

```

01 import tensorflow as tf
02 import matplotlib.pyplot as plt
03 import numpy as np
04 import os
05
06 from tensor2tensor import problems
07 from tensor2tensor.utils import trainer_lib
08 from tensor2tensor.utils import t2t_model
09 from tensor2tensor.utils import metrics
10
11 tfe = tf.contrib.eager
12 tf.enable_eager_execution() #启动动态图
13
14 problems.available() #显示T2T中的数据集

```

代码第 14 行列出了 T2T 框架中的所有数据集。

代码运行后，输出以下结果：

```
[ 'algorithmic_addition_binary40', # 算法数据集
  .....
  'algorithmic_sort_problem', # 语音数据集
  'audio_timit_characters_tune',
  .....
  'gym_simulated_discrete_problem_with_agent_on_wrapped_full_pong_autoencoded',
  'gym_wrapped_full_pong_random', # 强化学习相关数据集
  'image_celeba', # 图片数据集
  .....
  'image_mnist',
  'image_mnist_tune', # 语义数据集
  .....
  'img2img_imagenet',
  'lambada_lm',
  .....
  'languagemodel_wikitext103_characters',
  .....
  'translate_enzh_wmt8k',
  'video_bair_robot_pushing', # 视频数据集
  'video_bair_robot_pushing_with_actions',
  .....
  'wsj_parsing']
```

从上面结果可以看出，T2T 框架中的数据集几乎涵盖当今与深度学习相关的所有领域。

## 6.11.2 代码实现：构建 T2T 框架的工作路径及下载数据集

在 T2T 框架中有两个通用的文件目录用来管理数据集。

- tempdir：用于存放数据集原始文件。
- datadir：用于放置预处理之后的 TFRecord 格式文件。

下面按照指定路径建立文件目录并下载数据集。具体代码如下：

### 代码 6-20 在 T2T 框架中训练 mnist（续）

```
15 # 建立文件目录
16 data_dir = os.path.expanduser("./t2t/data")
17 tmp_dir = os.path.expanduser("./t2t/tmp")
18 tf.gfile.MakeDirs(data_dir)
19 tf.gfile.MakeDirs(tmp_dir)
20
21 # 下载数据集
22 mnist_problem = problems.problem("image_mnist")
23 mnist_problem.generate_data(data_dir, tmp_dir) # 下载，并拆分为训练和测试数据集，存到 data_dir 路径下
24
```

```

25 #取出一个数据并显示
26 Modes = tf.estimator.ModeKeys          #获取统一的数据集分类标志(用于测试或训练)
27 mnist_example = tfe.Iterator(mnist_problem.dataset(Modes.TRAIN,
28   data_dir)).next()                    #从训练集中取出一个元素
29 image = mnist_example["inputs"]        #一个数据集元素的张量
30 label = mnist_example["targets"]
31 plt.imshow(image.numpy()[:, :, 0].astype(np.float32),
32   cmap=plt.get_cmap('gray'))
33 print("Label: %d" % label.numpy())

```

代码第26行，使用了估算器中统一定义的数据集分类标志。该标志与T2T框架中拆分好的数据集相对应，其内部取值为（TRAIN ='train'、EVAL ='eval'、PREDICT ='infer'），即在代码第27行通过指定 Modes.TRAIN 便可以从训练数据集中取出数据。

整个代码运行后输出以下内容：

```
Label: 7
```

同时也输出了样本图片，如图6-9所示。

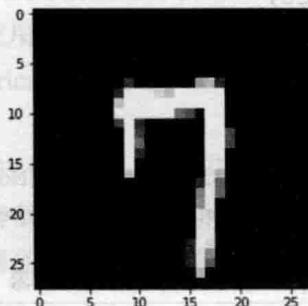


图6-9 MNIST数据集中label为7的样本

### 6.11.3 代码实现：在T2T框架中搭建自定义卷积网络模型

如想在T2T框架中使用自己的模型，则需要以下几个步骤。

- (1) 自定义模型：需要继承t2t\_model.T2TModel类，并实现body方法。
- (2) 为模型定义超参配置。
- (3) 定义损失函数集和优化器。

在具体实现时，定义MySimpleModel类继承于t2t\_model.T2TModel类。在MySimpleModel类的body方法中，用tf.layers接口定义了3个valid形式的卷积层。这3个卷积层将输入图片（尺寸为[28,28]）转换为向量特征（尺寸为[1,1]）。

具体代码如下：

#### 代码6-20 在T2T框架中训练mnist(续)

```

33 class MySimpleModel(t2t_model.T2TModel):      #自定义模型
34   pass

```

```

35     def body(self, features):
36         inputs = features["inputs"]
37         filters = self.hparams.hidden_size
38         #h1尺寸计算方法=(in_width-filter_width + 1) / strides_width =[12*12]
39         h1 = tf.layers.conv2d(inputs, filters, kernel_size=(5, 5), strides=(2,
2))#默认valid
40         #h2尺寸为 [4*4]
41         h2 = tf.layers.conv2d(tf.nn.relu(h1), filters, kernel_size=(5, 5),
strides=(2, 2))
42         #返回尺寸为[1*1]
43         return tf.layers.conv2d(tf.nn.relu(h2), filters, kernel_size=(3, 3))
44
45 hparams = trainer_lib.create_hparams("basic_1", data_dir=data_dir,
problem_name="image_mnist")
46 hparams.hidden_size = 64
47 model = MySimpleModel(hparams, Modes.TRAIN)
48
49 @tfe.implicit_value_and_gradients
50 def loss_fn(features): #用装饰器 implicit_value_and_gradients 封装 loss 函数
51     _, losses = model(features)
52     return losses["training"]
53
54 BATCH_SIZE = 128      #指定批次
55 #创建数据集
56 mnist_train_dataset = mnist_problem.dataset(Modes.TRAIN, data_dir)
57 mnist_train_dataset = mnist_train_dataset.repeat(None).batch(BATCH_SIZE)
58
59 optimizer = tf.train.AdamOptimizer()#定义优化器

```

在定义好模型 MySimpleModel 类之后，搭建反向结构的步骤如下：

(1) 用 trainer\_lib.create\_hparams 函数创建超参，并指定具体内容(hparams.hidden\_size = 64)，见代码第 45 行。

(2) 创建 loss 函数 loss\_fn，见代码第 50 行。

(3) 定义 Adam 优化器，见代码第 59 行。

其中，在第（2）步创建 loss 函数时，使用了 MySimpleModel 类的实例化对象。该对象会返回两个结果：模型的预测值和 loss 值。在 loss\_fn 函数中取出 loss 值，并忽略预测结果（见代码第 51 行）。

## 6.11.4 代码实现：用动态图方式训练自定义模型

在 T2T 框架中，训练自定义模型的方式与在动态图中训练模型的方式基本一致。具体代码如下：

### 代码 6-20 在 T2T 框架中训练 mnist (续)

```

60 NUM_STEPS = 500          #指定训练次数
61

```

```

62 for count, example in enumerate(mnist_train_dataset):
63     example["targets"] = tf.reshape(example["targets"], [BATCH_SIZE, 1, 1, 1])
#转为 4D
64     loss, gv = loss_fn(example)
65     optimizer.apply_gradients(gv)
66
67     if count % 50 == 0:
68         print("Step: %d, Loss: %.3f" % (count, loss.numpy()))
#输出训练过程中的 loss 值
69     if count >= NUM_STEPS:
70         break

```

代码第63行对标签做了形状变换。该标签用于计算loss值。因为自定义模型MySimpleModel输出的预测结果是一个形状为[BATCH\_SIZE, 1, 1, 1]的张量（见6.11.3小节），所以在计算loss值时，需要将标签转成同样形状的张量。

## 6.11.5 代码实现：在动态图中用metrics模块评估模型

TensorFlow中的metrics模块可以对模型进行自动评估。metrics模块是一个工具模块，可以非常方便地在动态图中被使用。使用方法具体分为三步：

- (1) 用metrics.create\_eager\_metrics方法创建一个metrics，返回两个函数metrics\_accum、metrics\_result。见代码第75行。
- (2) 用metrics\_accum函数计算评估结果。见代码第87行。
- (3) 用metrics\_result函数获取计算后的评估结果。见代码第89行。

从评估数据集里获取200个数据进行评估。具体代码如下：

代码6-20 在T2T框架中训练mnist（续）

```

71 model.set_mode(Modes.EVAL)
72 mnist_eval_dataset = mnist_problem.dataset(Modes.EVAL, data_dir) #定义评估
#数据集
73
74 #创建评估metrics，返回准确率
75 metrics_accum, metrics_result = metrics.create_eager_metrics(
76     [metrics.Metrics.ACC, metrics.Metrics.ACC_TOP5])
77
78 for count, example in enumerate(mnist_eval_dataset): #遍历数据
79     if count >= 200: #只取200个
80         break
81     example["inputs"] = tf.reshape(example["inputs"], [1, 28, 28, 1]) #变化形状
82     example["targets"] = tf.reshape(example["targets"], [1, 1, 1, 1])
83     predictions, _ = model(example) #用模型计算
84
85     metrics_accum(predictions, example["targets"]) #计算统计值
86
87

```

```

88 def body(self, metrics_result):
89     for name, val in metrics_result().items():
90         print("%s: %.2f" % (name, val))

```

代码运行后，输出以下结果：

```

Step: 0, Loss: 8.215
.....
Step: 500, Loss: 0.409
INFO:tensorflow:Setting T2TModel mode to 'eval'
.....
accuracy: 0.98
accuracy_top5: 1.00

```

## 6.12 实例 29：在 T2T 框架中，用自定义数据集训练中英文翻译模型

在 6.11 节中，实现了用 T2T 框架中的数据集训练自定义的模型。在实际应用中，更多的情况是——用自定义的数据集训练成熟的模型。

本实例将用自定义的中英文语料数据集训练 T2T 框架中的成熟模型，实现一个中英文翻译模型。

### 实例描述

有一个数据集，含有一万句中英文对应的平行语料。

要求用该数据集训练 T2T 框架中的成熟模型，使得模型能够用其他的样本完成翻译任务。



#### 提示：

平行语料是指，在中文、英文的两个数据文件中，每个文件中的样本都是按顺序一一对应的。

本实例使用了一个含有 10000 句平行语料的中英文样本集（在本书的配套资源中可以找到它）。在《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 9.8.6 小节的机器翻译实例中也使用过该样本集。

### 6.12.1 代码实现：声明自己的 problems 数据集

在 T2T 框架中制作 problems 数据集的步骤如下：

- (1) 单独创建一个 problem 文件夹。
- (2) 在 problem 文件夹下，创建代码文件“`__init__.py`”与“`my_problem.py`”。
- (3) 在代码文件“`__init__.py`”中添加代码，用于让系统自动加载“`my_problem.py`”代码文件。具体代码如下：

```
from . import my_problem
```

(4) 在代码文件“my\_problem.py”中定义 MyProblem 类，直接或间接继承于 problems 类。该类的名称必须与所在的代码文件名对应（对应规则为：将文件名“my\_problem.py”中的下划线去掉，分成两个单词，并将每个单词的首字母大写）。

(5) 用@registry.register\_problem 装饰器对 MyProblem 类进行修饰。该装饰器的作用是将数据集 MyProblem 类注册到 T2T 框架中。

具体代码如下：

#### 代码 my\_problem

```
01 from tensor2tensor.utils import registry
02 from tensor2tensor.data_generators import problem, text_problems
03
04 #自定义的problem一定要加该装饰器，否则t2t库找不到自定义的problem
05 @registry.register_problem
06 class MyProblem(text_problems.Text2TextProblem):
```

本实例使用的是文本类型的数据集，所以直接继承于 Text2TextProblem 类。

如果要使用其他类型的数据集，则需要在 T2T 框架的源码中查找对应类型的数据集 problem 类，并在自己的数据集类中添加继承关系。



#### 提示：

在 T2T 框架的源码中，数据集的源代码文件在 tensor2tensor\data\_generators 目录下。以作者本地路径为例，该文件的路径是：

C:\local\Anaconda3\lib\site-packages\tensor2tensor\data\_generators

### 6.12.2 代码实现：定义自己的 problems 数据集

下面按照 T2T 框架中规定的格式，在 MyProblem 类中实现 approx\_vocab\_size、is\_generate\_per\_split、dataset\_split、sgenerate\_samples 这几个方法。每个方法的作用见代码中的具体注释。

#### 代码 my\_problem（续）

```
07     @property
08     def approx_vocab_size(self):#指定词的个数
09         return 2**11
10
11     @property
12     def is_generate_per_split(self):
13         return False          #调用一次generate_samples，拆分数据集
14
15     @property
16     def dataset_splits(self):    #划分训练与评估数据集的比例
17         return [{               #可选参数
18             "split": problem.DatasetSplit.TRAIN,
```

```

19     "shards": 9,      # 在 model.txt 文中 "sharding": 9 表示将文本拆分 (9)
20   }, {
21     "split": problem.DatasetSplit.EVAL,
22     "shards": 1,
23   }]
24   #生成数据集
25   def generate_samples(self, data_dir, tmp_dir, dataset_split):
26     del data_dir
27     del tmp_dir
28     del dataset_split
29     #读取原始的训练样本数据
30     e_r = open(r"E:/t2t_test/tmp/english1w.txt", "r", encoding='utf-8')
31     c_r = open(r"E:/t2t_test/tmp/chinese1w.txt", "r", encoding='utf-8')
32
33     comment_list = e_r.readlines()
34     tag_list = c_r.readlines()
35     c_r.close()
36     e_r.close()
37     for comment, tag in zip(comment_list, tag_list):
38       comment = comment.strip()
39       tag = tag.strip()
40       yield {                         #返回样本与标签
41         "inputs": comment,
42         "targets": tag
43       }

```

代码第 12 行定义了方法 `is_generate_per_split`, 用于设置数据集的制作方式。

- 如果方法 `is_generate_per_split` 的返回值是 `True`, 则表示: 在进行训练集与评估集拆分时, 每次都需要调用 `generate_samples` 方法。
- 如果方法 `is_generate_per_split` 的返回值是 `False`, 则表示: 只用 `generate_samples` 方法将数据集解析一次, 然后进行拆分。

在实际使用中, 将方法 `is_generate_per_split` 的返回值设为 `False` 更为通用。

代码第 25 行定义了 `generate_samples` 方法, 用于生成数据。具体步骤如下:

- (1) 按照指定路径及读取方式读入样本数据。
- (2) 将读入的数据分成 `input` 与 `targets` 形式的字典对象 (见代码第 40 行)。



### 提示:

代码第 30 行中使用的是作者的本地样本路径 “E:/t2t\_test/tmp” 。在使用时, 读者可以根据自己的样本位置来修改路径。

## 6.12.3 在命令行下生成 TFRecord 格式的数据

下面以命令行方式调用 T2T 框架中的工具, 对文本进行预处理。以 Windows 系统为例, 具体步骤如下:

(1) 在 DOS 系统中, 通过 cd 命令来到本地 T2T 框架的安装路径 bin 下 (作者本地路径是: C:\local\Anaconda3\Lib\site-packages\tensor2tensor\bin)。

(2) 指定 t2t\_usr\_dir 参数为新建的 my\_problem.py 文件所在的路径 (作者的本地路径是: E:\t2t\_test\problem)。

(3) 指定 problem 参数为新建的 my\_problem.py 的文件名 “my\_problem”。

(4) 指定 data\_dir 路径为生成的 tfrecord 文件路径 (作者的本地路径是 E:\t2t\_test\data)。具体命令如下:

```
C:\local\Anaconda3\Lib\site-packages\tensor2tensor\bin>python      t2t_datagen.py
--t2t_usr_dir=E:\t2t_test\problem --problem=my_problem --data_dir=E:\t2t_test\data
```

执行该命令之后, 可以在本地“E:\t2t\_test\data”下看到生成的预处理文件及字典, 如图 6-10 所示。

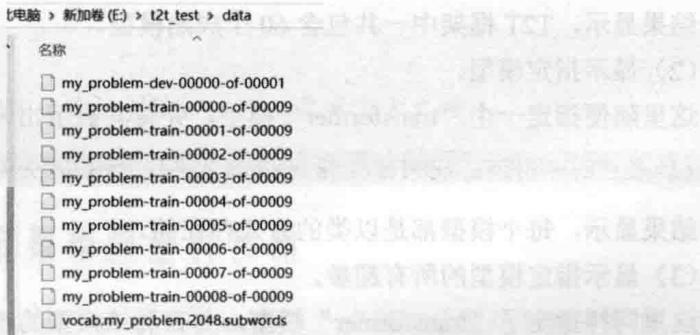


图 6-10 生成的预处理文件及字典

在图 6-10 中, 按照从上到下的顺序, 第 1 个文件是评估数据集, 最后一个文件是字典, 其他文件为训练数据集。

#### 6.12.4 查找 T2T 框架中的模型及超参, 并用指定的模型及超参进行训练

T2T 框架中内置了许多成熟模型及配套的超参。可以通过编写代码查看它们。

在得到可选的成熟模型及配套的超参之后, 便可以直接在命令行中指定自己的数据集, 并选取模型和超参进行训练, 不再需要额外编写代码。

##### 1. 在 T2T 中查找模型

编写代码查看 T2T 框架中的内置模型及对应超参。具体代码如下:

##### 代码 6-21 查看 T2T 模型及超参

```
01 import tensorflow as tf
02 from tensor2tensor import models
03
04 from tensor2tensor.utils import t2t_model
```

```

05 from tensor2tensor.utils import registry
06
07 print(len(registry.list_models()), registry.list_models()) #显示所有的模型
08 print(registry.model('transformer')) #显示指定模型
09 print(len(registry.list_hparams()), registry.list_hparams('transformer')) #显示指定模型的所有超参
10 print(registry.hparams('transformer_base_v1')) #显示指定模型的指定超参

```

代码运行后，输出以下结果：

(1) 显示所有的模型。

```

60 ['aligned', 'attention_lm', 'attention_lm_moe', 'autoencoder_autoregressive',
'autoencoder_basic', 'autoencoder_basic_discrete', ....,
'vqa_recurrent_self_attention', 'vqa_self_attention',
'vqa_simple_image_self_attention', 'xception']

```

结果显示，T2T 框架中一共包含 60 个成熟模型。

(2) 显示指定模型。

这里随便指定一个“transformer”模型，并将其显示出来（见代码第 8 行）。

```
<class 'tensor2tensor.models.transformer.Transformer'>
```

结果显示，每个模型都是以类的方式存在的。

(3) 显示指定模型的所有超参。

这里同样指定了“transformer”模型，并查看该模型的超参。

```

520 ['transformer_base_v1', 'transformer_base_v2', 'transformer_base',
'transformer_big',
....,
'transformer_symshard_base', 'transformer_symshard_sh4', 'transformer_symshard_lm_0',
'transformer_symshard_h4', 'transformer_teeny']

```

结果显示，T2T 框架中一共有 520 个超参数组合。它们都是已经微调过的超参数组合。用户直接拿来即可使用，非常方便。在这 520 个超参数组合中，可以找到关于“transformer”模型的超参数组合。

(4) 显示指定模型的指定超参数组合。

这里对“transformer”模型的“transformer\_base\_v1”超参数组合进行查看。

```

[('activation_dtype', 'float32'), ('attention_dropout', 0.0),
('attention_dropout_broadcast_dims', ''), ('attention_key_channels', 0),
('attention_value_channels', 0), ('attention_variables_3d', False), ('batch_size', 4096),
....,
('target_modality', 'default'), ('use_fixed_batch_size', False),
('use_pad_remover', True), ('use_target_space_embedding', True),
('video_num_input_frames', 1), ('video_num_target_frames', 1), ('vocab_divisor', 1),
('weight_decay', 0.0), ('weight_dtype', 'float32'), ('weight_noise', 0.0)]

```

从结果中可以看到，超参数组合里面放置了各个网络层的节点个数、优化算法等信息。

## 2. 在命令行中训练模型

下面在命令行中用 `t2t_trainer.py` 命令训练模型。这里指定模型为 `transformer`, 超参为 `transformer_base`。



**提示:**

如果本地机器只有一个 GPU, 则可以将超参换为 `transformer_base_single_gpu`。

具体代码如下:

```
C:\local\Anaconda3\Lib\site-packages\tensor2tensor\bin>python t2t_trainer.py
--t2t_usr_dir=E:\t2t_test\problem --problem=my_problem --data_dir=E:\t2t_test\data
--model=transformer --hparams_set=transformer_base --output_dir=E:\t2t_test\train
```

运行之后, 程序将循环训练模型, 并且每训练 1000 次模型之后保存一次检查点文件。



**提示:**

T2T 框架模型也支持分布式训练。具体训练方法可以参考官方文档:

[https://github.com/tensorflow/tensor2tensor/blob/master/docs/distributed\\_training.md](https://github.com/tensorflow/tensor2tensor/blob/master/docs/distributed_training.md)

## 6.12.5 用训练好的 T2T 框架模型进行预测

准备好一个英文文档 (路径是 `E:\t2t_test\decoder\en.txt`) , 在其中放置几个英语句子。通过在命令行里调用 T2T 框架中的 `t2t_decoder.py` 文件进行预测。具体命令如下:

```
C:\local\Anaconda3\Lib\site-packages\tensor2tensor\bin>python t2t_decoder.py
--t2t_usr_dir=E:\t2t_test\problem --problem=my_problem --data_dir=E:\t2t_test\data
--model=transformer --hparams_set=transformer_base --output_dir=E:\t2t_test\train
--decode_hparams="beam_size=4, alpha=0.6"
--decode_from_file=E:\t2t_test\decoder\en.txt
--decode_to_file=E:\t2t_test\decoder\ch.txt
```

本实例中, 使用了一个训练 2000 次的模型文件。运行后输出如下信息:

```
.....
INFO:tensorflow:Restoring parameters from E:\t2t_test\train\model.ckpt-2000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
.....
INFO:tensorflow:Inference results INPUT: to support its network information service ,
this super server is also installed with parallel network and e - mail service software ,
thus being able to support all kinds of popular database software .
INFO:tensorflow:Inference results OUTPUT: 这些 问题 , 可以 增加 信息 , 但 网络 网络 网络
网络 网络 网络 信息 , 网络 网络 网络 网络 网络 网络 网络 的 网络 网络 .
.....
INFO:tensorflow:Elapsed Time: 100.26935
INFO:tensorflow:Averaged Single Token Generation Time: 0.0214180
INFO:tensorflow:Writing decodes into E:\t2t_test\decoder\ch.txt
```

上面是训练 2000 次模型的预测结果，读者可以增加训练次数来达到更好的效果。运行之后，能够在 E:\t2t\_test\decoder\ch.txt 路径中找到模型的输出文件。



### 提示：

T2T 框架中默认的解码器只支持 UTF-8 格式。

如果是用 Windows 中新建的文本进行测试，还需要将其转为 UTF-8 格式，否则会报“UnicodeDecodeError: 'utf-8' codec”之类的错误信息。

将文本转换为 UTF-8 格式的方法有很多，比如：直接用编辑工具 UltraEdit 打开文本，然后单击菜单中的“文件”→“另存为”命令，选择编码为 UTF-8 格式。

## 6.12.6 扩展：在 T2T 框架中，如何选取合适的模型及超参

为了方便用户使用，在 T2T 框架的 GitHub 官网上给了一份详细的建议方案，针对不同的任务推荐不同的数据集、模型和超参，见表 6-2。

表 6-2 T2T 框架中不同任务的推荐训练方案

任 务	数据集	模型与超参
图像分类	ImageNet（一个大型数据集）：对应的 problem 为 image_imagenet，以及其重新缩放的版本（image_imagenet224、image_imagenet64、image_imagenet32）。 CIFAR-10：对应的 problem 为 image_cifar10，以及关闭数据扩充版本（image_cifar10_plain）。 MNIST：对应的 problem 为 image_mnist	ImageNet：建议使用 ResNet（对应的超参为 resnet_50）或 Xception 模型（对应的超参为 xception_base）。Resnet 应该在 ImageNet 上能够达到 76% 以上的准确率。 CIFAR 和 MNIST：建议使用 shake_shake 模型（对应的超参为 shakesshake_big）。经过 700000 次迭代训练后，该模型在 CIFAR-10 上可以达到接近 97% 的准确度
图像生成	CelebA：对应的 problem 为 img2img_celeba。用于图像到图像的转换，即从 8×8 pixel 到 32×32 pixel 的超分辨率。 CelebA-HQ：对应的 problem 为 image_celeba256_rev。 CIFAR-10：对应的 problem 为 image_cifar10_plain_gen_rev。用于生成 32×32 pixel 的条件分类任务。 LSUN Bedrooms：对应的 problem 为 image_lsun_bedrooms_rev。 MS-COCO：对应的 problem 为 image_text_ms_coco_rev。用于文本到图像的生成。 Small ImageNet（大型数据集）：ImageNet 的缩放版，分为 image_imagenet32_gen_rev 与 image_imagenet64_gen_rev 两个版本	建议使用 Image Transformer 模型（imagetransformer）或 Image Transformer plus 模型（imagetransformerpp）。 CIFAR-10：推荐使用的超参数集合为 imagetransformer_cifar10_base 或 imagetransformer_cifar10_base_dmol。 Imagenet-32：推荐使用的超参数集合为 imagetransformer_imagenet32_base

续表

任 务	数据集	模型与超参
语言建模	PTB（一个小数据集）：对应的 problem 为 languagemode_ptb10k（用于字级建模）和 languagemode_ptb_characters（用于字符级建模） LM1B（十亿字词语料库）：对应的 problem 为 languagemode_lm1b32k（用于字词级建模）和 languagemode_lm1b_characters（用于字符级建模）	建议使用 transformer 模型。 PTB：推荐使用超参 transformer_small。 LM1B：推荐使用超参 transformer_base
情绪分析	CNN / DailyMail：对应的 problem 为 summarize_cnn_dailymail32k	建议使用 transformer 模型（对应的超参为 transformer_prepend）
翻译	英语 - 德语：对应的 problem 为 translate_ende_wmt32k。 英语 - 法语：对应的 problem 为 translate_enfr_wmt32k。 英语 - 捷克语：对应的 problem 为 translate_encs_wmt32k。 英文 - 中文：对应的 problem 为 translate_enzh_wmt32k。 英语 - 越南语：对应的 problem 为 translate_envi_iwslt32k	建议使用 transformer 模型（对应的超参为 transformer_base）。 在单个 GPU 上，超参可使用 transformer_base_single_gpu。 在大型数据集上（例如 translate_enfr_wmt32k），超参可以使用 transformer_big

该建议方案支持的计算硬件为谷歌云 TPU 或带有 8 块 GPU 的机器。

更多的 T2T 框架示例，可以参考如下网址：

```
https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb
```

## 6.13 实例 30：将 TensorFlow 1.x 中的代码升级为可用于 2.x 版本的代码

在 TensorFlow 2.x 版本中，推荐使用估算器框架、动态图框架与 tf.keras 接口。1.x 版本中的静态图框架、tf-slim 接口将不再推荐使用。

在版本交替过程中，代码升级工作是避免不了的。本节将通过 `tf_upgrade_v2` 工具实现对已有代码的升级。

### 实例描述

将 6.3 节中在 TensorFlow 1.x 版本中编写的动态图代码，升级成符合 TensorFlow 2.x 版本语法的代码，并在 TensorFlow 2.x 版本中运行通过。

#### 6.13.1 准备工作：创建 Python 虚环境

本节的准备工作分为两部分。

(1) 安装 TensorFlow 2.x 版本：按照本书 2.6 节的内容，在本机创建虚拟环境，并安装 TensorFlow 2.x 版本。