深度学习之 TensorFlow

入门、原理与进阶实战

由工业机械出版社出版。预计 2018 年 1 月上市。敬请期待！

第 4 章　TensorFlow 编程基础

本章节主要学习 TensorFlow 的基础语法及功能函数。学完本章后，TensorFlow 代码对你来讲将不再陌生，你可以很轻易看懂网上和书中例子的代码，并可以尝试写一些简单的模型或算法。

学习一个开发环境应先从其内部入手，会起到一个事半功倍的效果。先从编程模型开始了解其运行机制，然后再介绍 TensorFlow 常用操作及功能函数，最后是共享变量、图和分布式部署。

本章含有教学视频，共 12 分 39 秒。

作者按照图书中本章的结构内容，对主要内容快速进行了讲解。特别是基本的模型工作机制、基础类型及操作、共享变量、图、分布式部署等内容（其中共享变量是本章的重点难点）

4.1　编程模型

TensorFlow 的命名来源于本身的运行原理。Tensor（张量）意味着 N维数组，Flow（流）意

味着基于数据流图的计算。TensorFlow 是张量从图像的一端流动到另一端的计算过程，这也是TensorFlow 的编程模型。

4.1.1 了解模型的运行机制

TensorFlow 的运行机制属于“定义”与“运行”相分离。从操作层面可以抽象成两种：模型构建和模型运行。

在模型构建过程中，需要先了解几个概念，见表 4-1。

表 4-1 模型构建中的概念

名称	含义
张量（tensor）	数据，即某一类型的多维数组
变量（Variable）	常用于定义模型中的参数，是通过不断的训练得到的值
占位符（placeholder）	输入变量的载体。也可以理解成定义函数时的参数
图中的节点操作（operation，OP）	即一个 OP 获得 0 个或多个 tensor,执行计算,输出额外的 0 个或多个 tensor

表 4-1 中定义的内容都是在一个叫做“图”的容器中完成的。关于“图”，有以下几点需要理解。

- 一个“图”代表一个计算任务。
- 在模型运行的环节中，“图”会在绘话（session）里被启动。
- session 将图的 OP 分发到诸如 CPU 或 GPU 之类的设备上，同时提供执行 OP 的方法。这些方法执行后,将产生的 tensor 返回。在 Python 语言中,返回的 tensor 是 numpy ndarray 对象；在 C 和 C++语言中,返回的 tensor 是 TensorFlow::Tensor 实例。

如图 4-1 表示了 session 与图的工作关系。

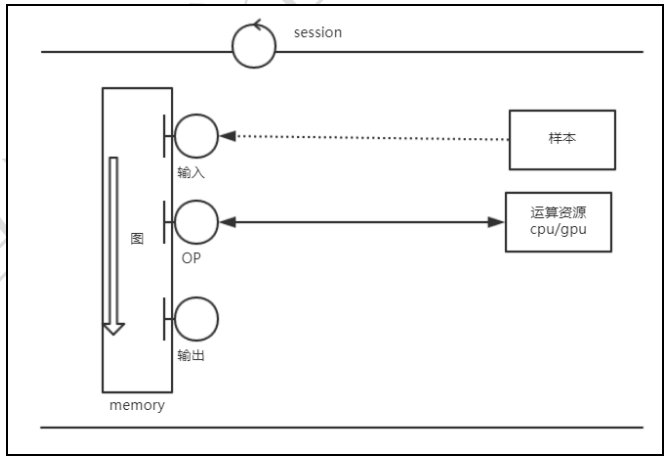

图 4-1 session 与图

在实际环境中，这种运行情况会有三种应用场景，训练场景、测试场景与使用场景。在训练场景下图的运行方式与其他两种不同，具体介绍如下：

（1）训练场景：主要是实现模型从无到有的过程，通过对样本的学习训练，调整学习参数，形成最终的模型。其过程是将给定的样本和标签作为输入节点，通过大量的循环迭代，将图中的

正向运算得到输出值，再进行反向运算更新模型中的学习参数。最终使模型产生的正向结果最大化的接近样本标签。这样就得到了一个可以拟合样本规律的模型。

（2）测试场景和使用场景：测试场景是利用图的正向运算得到结果比较与真实值的产别；使用场景也是利用图的正向运算得到结果，并直接使用。所以二者的运算过程是一样的。对于该场景下的模型与正常编程用到的函数特别相似。大家知道，在函数中，可以分为：实参、形参、函数体与返回值。同样在模型中，实参就是输入的样本，形参就是占位符，运算过程就相当于函数体，得到的结果相当于返回值。

另外 session 与图的交互过程中，还定义了两种数据的流向机制：

- 注入机制（feed）：通过占位符向模式中传入数据；
- 取回机制（fetch）：从模式中得到结果。

下面通过实例逐个演示 session 在各种情况下的用法。先从 session 的建立开始，接着演示 session 与图的交互机制，最后演示如何在 session 中指定 GPU 运算资源。

4.1.2 实例 5：编写 hello world 演示 session 的使用

先从一个 hello world 开始来理解 Session 的作用。

案例描述

建立一个 session，在 session 中输出 helloworld。

代码 4-1 sessionhello

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')  #定义一个常量
sess = tf.Session()                        #建立一个 session
print (sess.run(hello))                    #通过 session 里面的 run 来运行结果
sess.close()                               #关闭 session
```

运行代码 4-1 会得到如下输出：

```
b'Hello, TensorFlow!'
```

tf.constant 定义的是一个常量，hello 的内容只有在 session 的 run 内才可以返回。

可以试着在 2 和 3 行之间加入 print（hello）看一下效果，这时并不能输出 hello 的内容。

接下来换种写法，使用 with 语法来开启 session。

4.1.3 实例 6：演示 with session 的使用

With session 的用法是最常见的了，它沿用了 python 中 with 的语法，即当程序介绍后会自动关闭 session，而不需要在去写 close。代码如下。

案例描述

使用 with session 方法建立 session，并在 session 中计算两个变量（3 和 4）的相加与相乘。

代码 4-2 with session

```
import tensorflow as tf
a = tf.constant(3)                    #定义常量 3
b = tf.constant(4)                    #定义常量 4
with tf.Session() as sess:            #建立 session
```

```
print ("相加: %i" % sess.run(a+b))
print( "相乘: %i" % sess.run(a*b))
```

运行后得到如下输出：

```
相加: 7
相乘: 12
```

4.1.4 实例 7：演示注入机制

扩展上面代码：使用注入机制，将具体的实参注入到相应的 placeholder 中去。feed 只在调用它的方法内有效，方法结束后 feed 就会消失。

案例描述

定义占位符，使用 feed 机制将具体数值（3 和 4）通过占位符传入，并进行运算它们相加和相乘。

代码 4-3 withsessionfeed

```
01  import tensorflow as tf
02  a = tf.placeholder(tf.int16)
03  b = tf.placeholder(tf.int16)
04  add = tf.add(a, b)
05  mul = tf.multiply(a, b)                    #a 与 b 相乘
06  with tf.Session() as sess:
07      # 计算具体数值
08      print ("相加: %i" % sess.run(add, feed_dict={a: 3, b: 4}))
09      print ("相乘: %i" % sess.run(mul, feed_dict={a: 3, b: 4}))
```

运行代码，输出：

```
相加: 7
相乘: 12
```

标记的方法是：使用 tf.placeholder()为这些操作创建占位符，然后使用 feed_dict 来把具体的值放到站位符里。

```
注意：
关于 feed 中的 feed_dict 还有其他的方法，例如 update 等，在后文例子中用到时还会介绍。这里只是介绍最常用的方法。
```

4.1.5 建立 session 的其他方法

建立 session 还有两种方式：

- 交互式 session 方式：一般在 Jupiter 环境下使用较多，具体用法与前面的 with session 类似。代码如下：

```
sess = tf.InteractiveSession()
```

- 使用 Supervisor 方式：该方式会更为高级一些，使用起来也更加复杂，可以自动来管理 session 中的具体任务，比如，载入/载出检查点文件、写入 TensorBoard 等，另外还支持分布式训练的部署（在本书的后文会有介绍）。

4.1.6 实例 8：使用注入机制获取节点

在上例中，其实还可以一次将多个节点取出来。例如在最后一句加上如下代码：

案例描述

使用 fetch 机制将定义在图中的节点数值算出来。

代码 4-3　withsessionfeed（续）

```
10  ……
11  mul = tf.multiply(a, b)
12  with tf.Session() as sess:
13      # 将 op 运算通过 run 打印出来
14      print ("相加: %i" % sess.run(add, feed_dict={a: 3, b: 4}))#将 add 节点打印
    出来
15      print ("相乘: %i" % sess.run(mul, feed_dict={a: 3, b: 4}))
16      print (sess.run([mul, add], feed_dict={a: 3, b: 4}))
```

运行代码，输出：

```
相加: 7
相乘: 12
[12, 7]
```

4.1.7　指定 GPU 运算

如果下载的是 GPU 版本，在运行过程中 TensorFlow 能自动检测。如果检测到 GPU，TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作。

如果机器上有超过一个可用的 GPU，除第一个外的其他 GPU 默认是不参与计算的。为了让 TensorFlow 使用这些 GPU，必须将 OP 明确指派给它们执行。with……Device 语句用来指派特定的 CPU 或 GPU 执行操作：

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):
    a = tf.placeholder(tf.int16)
    b = tf.placeholder(tf.int16)
    add = tf.add(a, b)
   ……
```

设备用字符串进行标识。目前支持的设备包括：

- cpu:0：机器的 CPU。
- gpu:0：机器的第一个 GPU，如果有的话。
- gpu:1：机器的第二个 GPU，以此类推。

类似的还有通过 tf.ConfigProto 来构建一个 config，在 config 中指定相关的 gpu，并且在 session 中传入参数 config=“自己创建的 config”来指定 gpu 操作。

#tf.ConfigProto()的参数如下：

- log_device_placement=True：是否打印设备分配日志；
- allow_soft_placement=True：如果指定的设备不存在，允许 TF 自动分配设备。

使用举例：

```
config = tf.ConfigProto(log_device_placement=True,allow_soft_placement=True)
session = tf.Session(config=config, ...)
```

4.1.8 设置 GPU 使用资源

上文的 tf.ConfigProto()生成 config 之后，还可以设置其属性来分配 GPU 的运算资源。如下代码就是按需分配的意思：

```
config.gpu_options.allow_growth = True
```

使用 allow_growth option，刚一开始分配少量的 GPU 容量，然后按需慢慢的增加，由于不会释放内存，所以会导致碎片。

同样上述代码也可以放在 config 创建的时指定，举例：

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

如下代码还可以给 GPU 分配固定大小的计算资源。

```
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.7)
```

代表分配给 tensorflow 的 GPU 显存大小为：GPU 实际显存*0.7。

（该方法暂时用不到，读者在以后遇到这样的代码明白是什么意思就可以。）

4.1.9 保存和载入模型的方法介绍

一般而言，训练好的模型都是需要保存。这里将举例演示如何保存和载入模型。

1. 保存模型

首先需要建立一个 saver，然后在 session 中通过 saver 的 save 即可将模型保存起来。代码如下：

```
# 之前是各种构建模型 graph 的操作(矩阵相乘，sigmoid 等)
saver = tf.train.Saver()                                    # 生成 saver
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # 先对模型初始化
    # 然后将数据丢入模型进行训练 blablabla
    # 训练完以后，使用 saver.save 来保存
    saver.save(sess, "save_path/file_name")       #file_name 如果不存在，会自动创建
```

2. 载入模型

将模型保存好以后，载入也比较方便。在 session 中通过调用 saver 的 restore 函数，会从指定的路径找到模型文件，并覆盖到相关参数中。如下所示：

```
saver = tf.train.Saver()

with tf.Session() as sess:
    #参数可以进行初始化，也可不进行初始化。即使初始化了，初始化的值也会被 restore 的值给覆盖
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, "save_path/file_name") #会将已经保存的变量值 resotre 到 变量中。
```

4.1.10 实例 9：保存/载入线性回归模型

案例描述

在代码“3-1 线性回归.py”的基础上,添加模型的保存及载入功能。

通过扩展上一章的例子，来演示一下模型的保存及载入。在“代码 3-1 线性回归.py”中生成模拟数据之后，加入对图变量的重置，在 session 创建之前定义 saver 及保存路径，在 session 中训练结束后，保存模型。

代码 4-4　线性回归模型保存及载入

```
01  import tensorflow as tf
02  import numpy as np
03  import matplotlib.pyplot as plt
04
05  #模拟数据
06  ……
07  plt.plot(train_X, train_Y, 'ro', label='Original data')
08  plt.legend()
09  plt.show()
10
11  #重置图
12  tf.reset_default_graph()
13
14  #初始话等操作
15  ……
16  display_step = 2
17
18  saver = tf.train.Saver()                          # 生成 saver
19  savedir = "log/"                                  #生成模型的路径
20
21  # 启动 session
22  with tf.Session() as sess:
23      sess.run(init)
24      Sess 中的训练代码
25  ……
26      print (" Finished!")
27      saver.save(sess, savedir+"linermodel.cpkt") #保存模型
28      print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}), "W=",
    sess.run(W), "b=", sess.run(b))
29  #其他代码
30  ……
```

运行上面代码可以看到，在代码的同级目录下 log 文件夹里生成了几个文件，如图 4-2 所示。

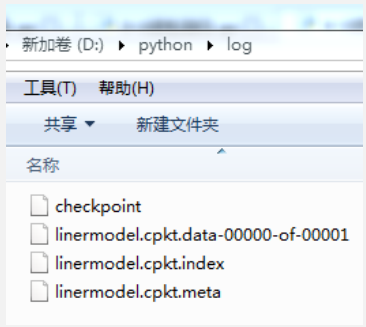

图 4-2　模型文件

再重启一个 session，并命名为 sess2，在代码里通过使用 saver 的 restore 函数，将模型载入。

代码 4-4　线性回归模型保存及载入（续）

```
with tf.Session() as sess2:
    sess2.run(tf.global_variables_initializer())
    saver.restore(sess2, savedir+"linermodel.cpkt")
    print ("x=0.2, z=", sess2.run(z, feed_dict={X: 0.2}))
```

为了测试效果，可以将前面一个 session 注释掉，运行之后可以看到如下输出：

```
INFO:tensorflow:Restoring parameters from log/linermodel.cpkt
x=0.2，z= [ 0.42615247]
```

表明模型已经成功载入，并计算出正确的值了。

4.1.11　实例 10：分析模型内容，演示模型的其他保存方法

下面再来详细介绍下关于模型保存的其他细节。

案例描述

将上一节生成的模型里面的内容打印出来，观察其存放的具体数据方式。同时演示将指定内容保存到模型文件中去。

1. 模型内容

虽然模型已经保存了。但是仍然对我们不透明。现在通过编写代码来将模型里面的内容打印出来，看看到底保存了哪些东西，都是什么样子。

代码 4-5　模型内容

```
from tensorflow.python.tools.inspect_checkpoint import print_tensors_in_checkpoint_file
savedir = "log/"
print_tensors_in_checkpoint_file(savedir+"linermodel.cpkt", None, True)
```

运行上面代码，打印如下信息：

```
tensor_name:  bias
[ 0.01919404]
tensor_name:  weight
[ 2.03479218]
```

可以看到，这个 tensor_name:后面跟的就是创建的变量名，接着是它的数值。

2. 保存模型的其他方法

前面的例子中 saver 的创建比较简单，其实 tf.train.Saver()里面开可以放参数来实现更高级的功能，可以指定存储变量名字与变量的对应关系。可以写成这样：

```
saver = tf.train.Saver({'weight': W, 'bias': b})
```

代表将 w 变量的值放到 weight 名字中去。类似的写法还有两种：

```
saver = tf.train.Saver([W, b]) # 放到一个 list 里.
```

```
saver = tf.train.Saver({v.op.name: v for v in [W, b]}) # 将 op 的名字当作 key:
```

下面扩展一下上述的例子，给 b 和 w 指定个固定值，并将他们颠倒放置。

代码 4-5　模型内容（续）

```
03  W = tf.Variable(1.0, name="weight")
04  b = tf.Variable(2.0, name="bias")
05
06  # 放到一个字典里：
07  saver = tf.train.Saver({'weight': b, 'bias': W})
08
09  with tf.Session() as sess:
10      tf.global_variables_initializer().run()
11      saver.save(sess, savedir+"linermodel.cpkt")
12
13  print_tensors_in_checkpoint_file(savedir+"linermodel.cpkt", None, True)
```

运行上面代码，输出如下信息：

```
tensor_name:  bias
1.0
tensor_name:  weight
2.0
```

例子中，W 值设为 1.0，b 的值设为 2.0。在创建 saver 时，我们将它们颠倒，可以看到保存的模型打印出来之后，bias 变成了 1.0，而 weight 变成了 2.0。

4.1.12　检查点（Checkpoint）

模型的保存场景并不限于在训练之后，在训练之中更需要保存，因为 TensorFlow 训练模型时难免会出现中断的情况。我们自然就希望能够将辛辛苦苦得到的中间参数保留下来，不然下次又要重新开始。

这种在训练中保存模型，习惯上称之为保存检查点。

4.1.13　实例 11：为模型添加保存检查点

案例描述

为一个线性回归任务的模型添加“保存检查点”功能。通过该功能，可以生成载入检查点文件，并能够指定生成检测点文件的个数。

与保存模型的功能类似，只不过是保存的位置发生了些变化，我们希望再显示信息的时将检查点保存起来，于是就放在了迭代训练中的打印信息后面。

另外，用到了 saver 的另一个参数——max_to_keep=1，表明最多只保存一个检查点文件。在保存时使用了如下代码，传入了迭代次数。

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

TensorFlow 会将迭代次数一起放在检查点的名字上，所以在载入时，同样也要指定迭代次数。

```
saver.restore(sess2, savedir+"linermodel.cpkt-" + str(load_epoch))
```

完整的代码如下：

代码 4-6　保存检查点

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

#定义生成 loss 可视化的函数
plotdata = { "batchsize":[], "loss":[] }
def moving_average(a, w=10):
    if len(a) < w:
        return a[:]
    return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in enumerate(a)]

#生成模拟数据
train_X = np.linspace(-1, 1, 100)
train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 # y=2x, 但是加入了
噪声
#图形显示
plt.plot(train_X, train_Y, 'ro', label='Original data')
plt.legend()
plt.show()

tf.reset_default_graph()

# 创建模型
# 占位符
X = tf.placeholder("float")
Y = tf.placeholder("float")
# 模型参数
W = tf.Variable(tf.random_normal([1]), name="weight")
b = tf.Variable(tf.zeros([1]), name="bias")
# 前向结构
z = tf.multiply(X, W)+ b

#反向优化
cost =tf.reduce_mean( tf.square(Y - z))
learning_rate = 0.01
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) #
梯度下降

# 初始化所有变量
init = tf.global_variables_initializer()
# 定义学习参数
training_epochs = 20
display_step = 2
saver = tf.train.Saver(max_to_keep=1) # 生成 saver
savedir = "log/"
# 启动图
with tf.Session() as sess:
    sess.run(init)

    # 向模型输入数据
    for epoch in range(training_epochs):
```

```
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        #显示训练中的详细信息
        if epoch % display_step == 0:
            loss = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
            print ("Epoch:", epoch+1, "cost=", loss,"W=", sess.run(W), "b=",
sess.run(b))
            if not (loss == "NA" ):
                plotdata["batchsize"].append(epoch)
                plotdata["loss"].append(loss)
            saver.save(sess, savedir+"linermodel.cpkt", global_step=epoch)

    print (" Finished!")

    print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}), "W=",
sess.run(W), "b=", sess.run(b))

    #显示模型
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()

    plotdata["avgloss"] = moving_average(plotdata["loss"])
    plt.figure(1)
    plt.subplot(211)
    plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
    plt.xlabel('Minibatch number')
    plt.ylabel('Loss')
    plt.title('Minibatch run vs. Training loss')

    plt.show()

#重启一个 session ，载入检查点
load_epoch=18
with tf.Session() as sess2:
    sess2.run(tf.global_variables_initializer())
    saver.restore(sess2, savedir+"linermodel.cpkt-" + str(load_epoch))
    print ("x=0.2, z=", sess2.run(z, feed_dict={X: 0.2}))
```

上面代码运行完后，会看到在 log 文件夹下多了几个 linermodel.cpkt-18*的文件。它就是检查点文件。

这里面使用 tf.train.Saver(max_to_keep=1)代码创建 saver 时传入的参数 max_to_keep=1 代表：在迭代过程中只保存一个文件。这样，在循环训练过程中，新生成的模型就会覆盖掉以前的模型。

注意：

如果觉得通过指定迭代次数比较麻烦，还有一个好方法，可以快速获取到检查点文件。示例代码如下：

```
ckpt = tf.train.get_checkpoint_state(ckpt_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)

还可以再简洁一些，写成如下的样子：
kpt = tf.train.latest_checkpoint(savedir)
    if kpt!=None:
        saver.restore(sess, kpt)
```

4.1.14 实例 12：更简便地保存检查点

本例中介绍另一种更简便地保存检查点功能代码的方法——tf.train.MonitoredTrainingSession 函数。该函数可以直接实现保存及载入检查点模型的文件。与前面的方式不同，并不是按照循环步数来保存的，而是按照训练时间。通过指定 save_checkpoint_secs 参数的具体秒数，来设置每训练多久保存一次检查点。

案例描述

演示使用 MonitoredTrainingSession 函数来自动管理检查点文件。

具体代码如下：

代码 4-7 trainMonitored

```
import tensorflow as tf
tf.reset_default_graph()
global_step = tf.contrib.framework.get_or_create_global_step()
step = tf.assign_add(global_step, 1)
#设置检查点路径为 log/checkpoints
with
tf.train.MonitoredTrainingSession(checkpoint_dir='log/checkpoints',save_check
point_secs  = 2) as sess:
   print(sess.run([global_step]))
   while not sess.should_stop():#启用死循环，当 sess 不结束时就不停止。
      i = sess.run( step)
      print( i)
```

运行代码，得到如下输出：

```
252 12851
252 12852
252 12853
252 12854
252 12855
252 12856
```

将程序停止，可以看到“log/checkpoints”下面生成了检测点文件“model.ckpt-8968.meta ”。再次运行，输出如下：

```
252 8969
252 8970
```

```
252 8971
```

可见，程序自动载入检查点文件是从第 8969 次开始运行。

注意：

（1） 如果不设置 save_checkpoint_secs 参数，默认的保存时间间隔为 10 分钟。这种按照时间保存的模式更适用于使用大型数据集来训练复杂模型的情况。

（2） 使用该方法时，必须要定义 global_step 变量，否则回报错误。

4.1.15 模型操作常用函数总结

这里将模型操作的相关函数来个系统的介绍见表 4-2。

表 4-2 模型操作相关函数

函数	说明
tf.train.Saver (var_list=None, reshape=False, sharded=False, max_to_keep=5, keep_checkpoint_every_n_hours=10000.0, name=None, restore_sequentially=False, saver_def=None, builder=None)	创建存储器 Saver
tf.train.Saver.save(sess, save_path, global_step=None, latest_filename=None, meta_graph_suffix=’meta’, write_meta_graph=True)	保存
tf.train.Saver.restore(sess, save_path)	恢复
tf.train.Saver.last_checkpoints	列出最近未删除的 checkpoint 文件名
tf.train.Saver.set_last_checkpoints(last_checkpoints)	设置 checkpoint 文件名列表
tf.train.Saver.set_last_checkpoints_with_time(last_checkpoints_with_time)	设置 checkpoint 文件名列表和时间戳

4.1.16 TensorBoard 可视化介绍

TensorFlow 还提供了一个可视化工具——TensorBoard。它可以将训练过程中的各种绘制数据展示出来，包括标量（scalars）、图片（images）、音频（Audio）、计算图（graph）、数据分布、直方图（histograms）和嵌入式向量。可以通过网页来观察模型的结构和训练过程中各个参数的变化。

当然，TensorBoard 不会自动把代码展示出来，其实它是一个日志展示系统，需要在 session 中运算图时，将各种类型的数据汇总并输出到日志文件中。然后启动 TensorBoard 服务，TensorBoard 读取这些日志文件，并开启 6006 端口提供 web 服务，让我们可以在浏览器中查看数据。

TensorFlow 提供了一系列 API 来生成这些数据具体见表 4-3。

表 4-3 模型操作相关函数

函数	说明
tf.summary.scalar(tags, values, collections=None, name=None)	标量数据汇总，输出 protobuf
tf.summary.histogram(tag, values, collections=None, name=None)	记录变量 var 的直方图，输出带直方图的汇总的 protobuf

tf.summary.image(tag, tensor, max_images=3, collections=None, name=None)	图像数据汇总，输出 protobuf
tf.summary.merge(inputs, collections=None, name=None)	合并所有汇总
tf.summary.FileWriter	创建一个 SummaryWriter
Class SummaryWriter: add_summary(), add sessionlog(), add_event(), or add_graph()	将 protobuf 写入文件的类

4.1.17 实例 13：线性回归的 TensorBoard 可视化

下面举例来演示一下 TensorBoard 的可视化效果。

案例描述

为“3-1 线性回归.py”代码文件添加支持输出 TensorBoard 信息的功能，演示通过 TensorBoard 来观察训练过程。

还是以“3-1 线性回归.py”的代码为原型，在上面添加支持 TensorBoard 的功能。该例子中，通过添加一个标量数据和一个直方图数据到 log 里，然后通过 TensorBoard 显示出来。代码改动量非常小，第一步加入到 summary，第二步写入文件。

将模型的生成值加入到直方图数据中，将损失值加入到标量数据中，代码如下：

代码 4-8 线性回归的 TensorBoard 可视化

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

……
# 前向结构
z = tf.multiply(X, W)+ b
tf.summary.histogram('z',z)                    #将预测值以直方图显示
#反向优化
cost =tf.reduce_mean( tf.square(Y - z))
tf.summary.scalar('loss_function', cost)  #将损失以标量显示
……
给直方图起名仍然叫 z，标量的名字叫 loss_function。
下面的代码是在启动 session 之后加入代码，创建一个 summary_writer，在迭代中将 summary 的值运行生成出来，同时添加到文件里。
# 启动 session
with tf.Session() as sess:
    sess.run(init)

    merged_summary_op = tf.summary.merge_all()#合并所有 summary
    #创建 summary_writer，用于写文件
    summary_writer                                                    =
tf.summary.FileWriter('log/mnist_with_summaries',sess.graph)

    # 向模型输入数据
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})
```

```
        #生成 summary
        summary_str = sess.run(merged_summary_op,feed_dict={X: x, Y: y});
        summary_writer.add_summary(summary_str, epoch);#将 summary 写入文件

    ……
```

运行一下，显示的东西还和以前一样没什么变化，来到生成的路径下可以看到多了一个文件，如图 4-3 所示。

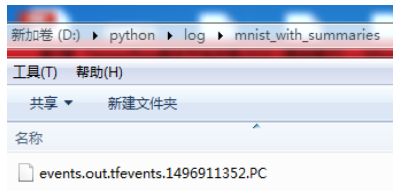

图 4-3 summary 文件

然后单击“开始”菜单/运行/输入“cmd”，启动“命令行”窗口，首先来到 summary 日志的上级路径下，输入命令：

```
tensorboard --logdir D:\python\log/mnist_with_summaries
```

结果如图 4-4 所示。

```
C:\Users\Administrator>d:

D:\>cd python

D:\python>cd log

D:\python\log>tensorboard --logdir D:\python\log/mnist_with_summaries
Starting TensorBoard b'54' at http://PC:6006
(Press CTRL+C to quit)
```

图 4-4 启动 TensorBoard

接着打开 chrome 浏览器，输入 http://127.0.0.1:6006，会看到如图 4-5 所示界面。单击上面的 scalars，会看到之前创建的 loss_fuction。这个 loss_fuction 也是可以点的，打开后就是损失值随迭代次数的变化。

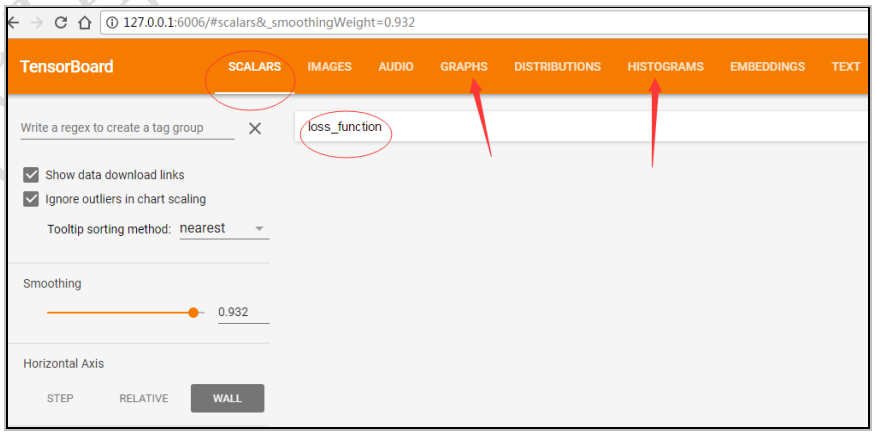

图 4-5 TensorBoard 界面

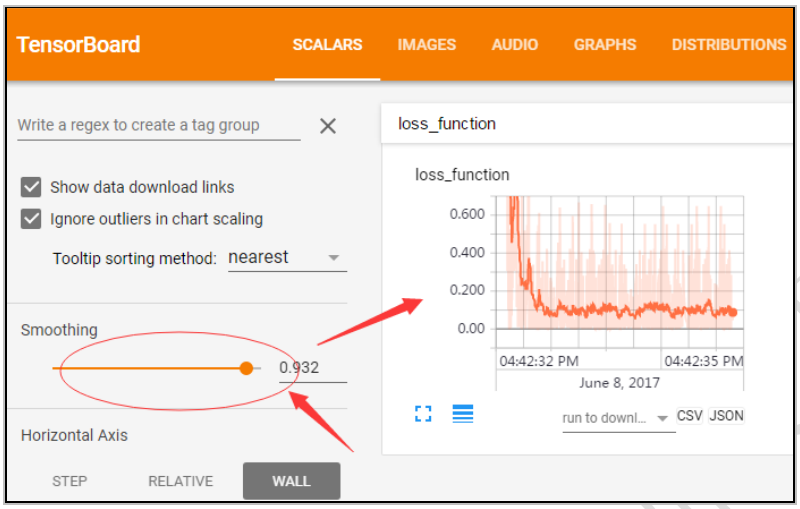

图 4-6 TensorBoard 标量

在图 4-6 中可以调节平滑数来改变右边标量的曲线。类似的还可以点开图 4-5 中的 Graphs 看看神经网络的内部结构，还可以点开图 4-5 中的 HISTOGRAMS 来看例子中的另一个显示值 z。

注意：

在显示 TensorBoard 界面的过程中，下面两点需要强调一下：

- 浏览器最好要使用 chrome。
- 在命令行里启动 TensorBoard 时，一定要先进入到日志所在的上级路径下。否则打开的页面里找不到创建好的信息。

4.2 TensorFlow 基础类型定义及操作函数介绍

下面介绍 TensorFlow 的基础类型、基础函数。这部分学完，会使你对 TensorFlow 的基础语法有个系统的了解，为后面学习写代码或读代码扫清障碍。

4.2.1 张量及操作

张量可以说是 TensorFlow 的标志，因为整个框架的名称 TensorFlow 就是张量流的意思。下面来一起全面的认识一下张量。

1. 张量介绍

TensorFlow 程序使用 tensor 数据结构来代表所有的数据。计算图中，操作间传递的数据都是 tensor。

可以把 TensorFlow tensor 看作是一个 n 维的数组或列表。每个 tensor 包含了类型（type）、阶（rank）和形状（shape）。

- Tensor 类型

为了方便理解，这里将 tensor 的类型与 Python 的类型放在一起做了个比较，见表 4-2。

表 4-2 张量类型

数据类型	Python 类型	描述
DT_FLOAT	tf.float32	32 位浮点数.
DT_DOUBLE	tf.float64	64 位浮点数.
DT_INT64	tf.int64	64 位有符号整型.
DT_INT32	tf.int32	32 位有符号整型
DT_INT16	tf.int16	16 位有符号整型
DT_INT8	tf.int8	8 位有符号整型
DT_UINT8	tf.uint8	8 位无符号整型
DT_STRING	tf.string	可变长度的字节数组.每一个张量元素都是一个字节数组.
DT_BOOL	tf.bool	布尔型
DT_COMPLEX64	tf.complex64	由两个 32 位浮点数组成的复数:实数和虚数.
DT_QINT32	tf.qint32	用于量化 Ops 的 32 位有符号整型
DT_QINT8	tf.qint8	用于量化 Ops 的 8 位有符号整型
DT_QUINT8	tf.quint8	用于量化 Ops 的 8 位无符号整型

- rank（阶）：

rank（阶）指的就是维度。但张量的阶和矩阵的阶并不是同一个概念，主要是看有几层中括号。比如，对于一个传统意义上的 3 阶矩阵 a=[[1,2,3],[4,5,6],[7,8,9]]来讲，在张量中的阶数表示为 2 阶。（因为它有两层中括号）

表 4-3 列出了标量、向量、矩阵的阶数。

表 4-3 标量向量和矩阵的阶数

rank	实例	例子
0	标量(只有大小)	a = 1
1	向量(大小和方向)	b = [1,1,1,1]
2	矩阵(数据表)	C =[[1,1],[1,1]]
3	3 阶张量 (数据立体)	D =[[[1],[1]],[[1],[1]]]
n	n 阶	E = [[[[[[…[[1],[1],]]]…]]]]](n 层中括号)

- shape（形状）

shape（形状）用于描述张量内部的组织关系。“形状”可以通过 Python 中的整数列表或元组（int list 或 tuples）来表示，也或者用 TensorFlow 中的相关形状函数来表示。

举例：一个二阶张量 a = [[1, 2, 3], [4, 5, 6]]形状是两行三列，描述为（2,3)。

2. 张量相关操作

张量的相关操作包括类型转换、数值操作、形状变换和数据操作。

- 类型转换

类型转换的相关函数见表 4-4。

表 4-4 类型变换相关函数

函数	描述
tf.string_to_number (string_tensor, out_type=None, name=None)	字符串转为数字
tf.to_double(x, name=’ToDouble’)	转为 64 位浮点类型——之后 x 变为 float64 类型
tf.to_float(x, name=’ToFloat’)	转为 32 位浮点类型——之后 x 变为 float32 类型
tf.to_int32(x, name=’ToInt32’)	转为 32 位整型——之后 x 变为 int32 类型
tf.to_int64(x, name=’ToInt64’)	转为 64 位整型——之后 x 变为 int64 类型
tf.cast(x, dtype, name=None)	将 x 或者 x.values 转换为 dtype 所指定的类型 W = tf.Variable(1.0) tf.cast(W, tf.int32) ==> W=1 # dtype=tf.int32

- 数值操作

数值操作的相关函数见表 4-5。

表 4-5 类型变换相关函数

函数	描述
tf.ones(shape,dtype)	按指定类型与形状生成值为 1 的张量 tf.ones([2, 3], tf.int32)==> [[1 1 1] [1 1 1]]
tf.zeros(shape,dtype)	按指定类型与形状生成值为 0 的张量 tf.ones([2, 3], tf.int32)==> [[0 0 0] [0 0 0]] 按上一行的改。
tf.ones_like（input）	生成输入张量一样形状和类型的 1 tensor=[[1, 2, 3], [4, 5, 6]] tf.ones_like（tensor）==> [[1 1 1] [1 1 1]]
tf.zeros_like（input）	生成输入张量一样形状和类型的 1 tensor=[[1, 2, 3], [4, 5, 6]] tf.zeros_like（tensor）==>[[0 0 0] [0 0 0]]
tf.fill(shape,value)	为指定形状填值 tf.fill([2,3],1) ==> [[1 1 1] [1 1 1]]
tf.constant(value, shape)	生成常量 tf.constant(1 ,[2,3]) ==> [[1 1 1] [1 1 1]]
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	正太分布随机数，均值 mean,标准差 stddev
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	截断正态分布随机数，均值 mean,标准差 stddev,不过只保留[mean-2*stddev,mean+2*stddev]范围内的随机数
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)	均匀分布随机数，范围为[minval,maxval]
tf.random_crop(value, size, seed=None, name=None)	将输入值 value 按照 size 尺寸随机剪辑
tf.set_random_seed(seed)	设置随机数种子
tf.linspace(start,stop,num,name=None)	在[start,stop]范围内产生 num 个数的等差数列。不过注意，start 和 stop 要用浮点数表示，不然会报错

	tf.linspace(start=1.0,stop=5.0,num=5,name=None) [1. 2. 3. 4. 5.]
tf.range(start,limit=None,delta=1,name=' range')	在[start,limit)范围内以步进值 delta 产生等差数列。注意，是不包括 limit 在内的 tf.range(start=1,limit=5,delta=1) [1 2 3 4]

- 形状变换

形状变换的相关函数见表 4-6。

表 4-6 形状变换相关函数

函数	描述
tf.shape(input, name=None)	返回数据的 shape # 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] shape(t) ==> [2, 2, 3]
tf.size(input, name=None)	返回数据的元素数量 # 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]] size(t) ==> 12
tf.rank(input, name=None)	返回 tensor 的 rank 注意：此 rank 不同于矩阵的 rank， tensor 的 rank 表示一个 tensor 需要的索引数目来唯一表示任何一个元素 也就是"order", "degree"或"ndims" #' t' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] # shape of tensor 't' is [2, 2, 3] rank(t) ==> 3
tf.reshape(tensor, shape, name=None)	改变 tensor 的形状 # tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9] # tensor 't' has shape [9] reshape(t, [3, 3]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9]] #如果 shape 有元素[-1],表示在该维度打平至一维 # -1 将自动推导得为 9: reshape(t, [2, -1]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3], [4, 4, 4, 5, 5, 5, 6, 6, 6]]

tf.expand_dims(input, dim, name=None)	插入维度 1 进入一个 tensor 中
	#该操作要求-1-input.dims()
	t =[[2,3,3],[1,5,5]]
	t1 = tf.expand_dims(t, 0)
	t2 = tf.expand_dims(t, 1)
	t3 = tf.expand_dims(t, 2)

	t4 = tf.expand_dims(t, -1)
	#t4 = tf.expand_dims(t, 3)#出错
	print(np.shape(t))#(2, 3)
	print(np.shape(t1))#(1, 2, 3)
	print(np.shape(t2))#(2, 1, 3)
	print(np.shape(t3))#(2, 3, 1)
	print(np.shape(t4))#(2, 3, 1)

tf.squeeze(input, dim, name=None)	将 dim 指定的为 1 的维度去掉（如果不为 1，会报错）
	t =[[[[2],[1]]]]
	t1 = tf.squeeze(t, 0)
	t2 = tf.squeeze(t, 1)
	t3 = tf.squeeze(t, 3)
	t4 = tf.squeeze(t, -1)
	#t4 = tf.squeeze(t, 2)#出错,因为 2 对应的维度为 2
	print(np.shape(t))#(1, 1, 2, 1)
	print(np.shape(t1))#(1, 2, 1)
	print(np.shape(t2))#(1, 2, 1)
	print(np.shape(t3))#(1, 1, 2)
	print(np.shape(t4))#(1, 1, 2)

- 数据操作

数据操作的相关函数见表 4-7：

表 4-7 数据操作相关函数

函数	描述
tf.slice(input_, begin, size, name=None)	对 tensor 进行切片操作。 其中 size[i] = input.dim_size(i) - begin[i] 该操作要求 0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n] #’input’ is #[[[1, 1, 1], [2, 2, 2]],[[3, 3, 3], [4, 4, 4]],[[5, 5, 5], [6, 6, 6]]] tf.slice(input, [1, 0, 0], [1, 1, 3]) ==> [[[3, 3, 3]]] tf.slice(input, [1, 0, 0], [1, 2, 3]) ==> [[[3, 3, 3], [4, 4, 4]]] tf.slice(input, [1, 0, 0], [2, 1, 3]) ==> [[[3, 3, 3]], [[5, 5, 5]]]
tf.split(value, num_or_size_splits, axis=0, num=None, name="split")	沿着某一维度将 tensor 分离为 num_or_size_splits # ’value’ is a tensor with shape [5, 30] # 沿着第一列将 value 按[4, 15, 11]分成 3 个张量 split0, split1, split2 = tf.split(value, [4, 15, 11], 1) tf.shape(split0) ==> [5, 4] tf.shape(split1) ==> [5, 15] tf.shape(split2) ==> [5, 11]
tf.concat(concat_dim, values,	沿着某一维度连结 tensor

name=’ concat’)	t1 = [[1, 2, 3], [4, 5, 6]] t2 = [[7, 8, 9], [10, 11, 12]] tf.concat([t1, t2], 0) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]] tf.concat([t1, t2],1) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]] 如果想沿着 tensor 一新轴连结打包,那么可以: tf.concat(axis, [tf.expand_dims(t, axis) for t in tensors]) 等同于 tf. stack (tensors, axis=axis)
tf.pack(values, axis=0, name=’ pack’) tf.unpack	将一个 R 维张量列表沿着 axis 轴组合成一个 R+1 维的张量 Pack 和 unpack 在 TensorFlow1.0 版本之后已经被 stack 和 unstack 代替
tf.stack(input, axis=0)	tf.stack 将一个 R 维张量列表沿着 axis 轴组合成一个 R+1 维的张量 tensor=[[1, 2, 3], [4, 5, 6]] tensor2=[[10, 20, 30],[40, 50, 60]] tf.stack([tensor, tensor2]) =>[[[1 2 3] [4 5 6]] [[10 20 30][40 50 60]]] tf.stack([tensor, tensor2], axis=1) =>[[[1 2 3][10 20 30]] [[4 5 6][40 50 60]]]
def unstack(value, num=None, axis=0, name="unstack")	unstack 拆分行列 axis=0 按行拆。 axis=1 按列拆，num 为输出 list 的个数，必须与预计输出的相等，不然会报错，可忽略这个参数。 tensor=[[1, 2, 3], [4, 5, 6]] tf.unstack(tensor) => [array([1, 2, 3]), array([4, 5, 6])] tf.unstack(tensor,axis=1) =>[array([1, 4]), array([2, 5]), array([3, 6])] #tensor.shape=[2,3],axis=0,就是分成 2 个。axis=1 就是分成 3 个 #ten2.shape=[2,3,4],axit=2 就是分成 4 个
tf.reverse(tensor, dims, name=None)	沿着某维度进行序列反转 其中 dim 为列表，元素为 bool 型，size 等于 rank(tensor) # tensor ‘t’ is [[[[0, 1, 2, 3], #[4, 5, 6, 7], #[8, 9, 10, 11]], #[[12, 13, 14, 15], #[16, 17, 18, 19], #[20, 21, 22, 23]]]] # tensor ‘t’ shape is [1, 2, 3, 4] # ‘dims’ is [False, False, False, True] reverse(t, dims) ==> [[[[3, 2, 1, 0], [7, 6, 5, 4], [11, 10, 9, 8]],

	[[15, 14, 13, 12], [19, 18, 17, 16], [23, 22, 21, 20]]]]
tf.transpose(a, perm=None, name=’transpose’)	调换 tensor 的维度顺序 按照列表 perm 的维度排列调换 tensor 顺序， 如为定义，则 perm 为(n-1…0) # ‘x’ is [[1 2 3],[4 5 6]] tf.transpose(x) ==> [[1 4], [2 5],[3 6]] # Equivalently tf.transpose(x, perm=[1, 0]) ==> [[1 4],[2 5], [3 6]]
tf.gather(params, indices, validate_indices=None, name=None)	合并索引 indices 所指示 params 中的切片 y= tf.constant([0.,2.,-1.]) t = tf.gather(y, [2,0]) sess=tf.Session() t2 = sess.run([t]) print(t2)---》[array([-1., 0.], dtype=float32)]
tf.one_hot (indices, depth, on_value=None, off_value=None, axis=None, dtype=None, name=None)	indices = [0, 2, -1, 1] depth = 3 on_value = 5.0 off_value = 0.0 axis = -1 #Then output is [4 x 3]: output = [5.0 0.0 0.0] // one_hot(0) [0.0 0.0 5.0] // one_hot(2) [0.0 0.0 0.0] // one_hot(-1) [0.0 5.0 0.0] // one_hot(1)
tf.count_nonzero (input_tensor, axis=None, keep_dims=False, dtype=dtypes.int64, name=None, reduction_indices=None)	#统计非 0 个数

注意：

tf 开头的代码都不能直接运行，必须放到 session 里面才可以。举例：

```
01  import numpy as np
02  import tensorflow as tf
03
04  x = tf.constant(2)
05  y = tf.constant(5)
06  def f1(): return tf.multiply(x, 17)
07  def f2(): return tf.add(y, 23)
08  r = tf.cond(tf.less(x, y), f1, f2)
09  print(r)#这样是错的
10
11  #生成 2 行三列的张量，值为 1
12  with tf.Session() as sess:
13      print(sess.run( r  ))#这样才可以
```

4.2.2 算术运算函数

表 4-8 列出了 TensorFlow 关于算术运算方面的函数。

表 4-8 算术操作

函数	描述
tf.assign(x, y, name=None)	令 x=y
tf.add(x, y, name=None)	求和
tf.subtract (x, y, name=None)	减法
tf.multiply (x, y, name=None)	乘法
tf.divide (x, y, name=None)	除法,也可以 tf.div
tf.mod(x, y, name=None)	取模
tf.abs(x, name=None)	求绝对值
tf.negative(x, name=None)	取负 (y = -x).
tf.sign(x, name=None)	返回符号 y = sign(x) = -1 if x < 0; 0 if x == 0; 1 if x > 0.
tf.inv(x, name=None)	取反
tf.square(x, name=None)	计算平方 (y = x * x = x^2).
tf.round(x, name=None)	舍入最接近的整数 # 'a' is [0.9, 2.5, 2.3, 1.5, -4.5] tf.round(a) ==> [1.0, 2.0, 2.0, 2.0, -4.0] #如果需要真正的四舍五入，可以用 tf.int32 类型强转
tf.sqrt(x, name=None)	开根号 (y = \sqrt{x} = x^{1/2}).
tf.pow(x, y, name=None)	幂次方 # tensor 'x' is [[2, 2], [3, 3]] # tensor 'y' is [[8, 16], [2, 3]] tf.pow(x, y) ==> [[256, 65536], [9, 27]]
tf.exp(x, name=None)	计算 e 的次方
tf.log(x, name=None)	计算 log，一个输入计算 e 的 ln，两输入以第二输入为底
tf.maximum(x, y, name=None)	返回最大值 (x > y ? x : y)
tf.minimum(x, y, name=None)	返回最小值 (x < y ? x : y)
tf.cos(x, name=None)	三角函数 cosine
tf.sin(x, name=None)	三角函数 sine
tf.tan(x, name=None)	三角函数 tan
tf.atan(x, name=None)	三角函数 ctan

tf. cond(pred, true_fn=None, false_fn=None, strict=False, name=None,
fn1=None, fn2=None)
	#满足条件就执行 fn1,否则执行 fn2
	x = tf.constant(2)
	y = tf.constant(5)
	def f1(): return tf.multiply(x, 17)
	def f2(): return tf.add(y, 23)
	r = tf.cond(tf.less(x, y), f1, f2)
	=> 34

4.2.3 矩阵相关的运算

矩阵相关的操作函数见表 4-9。

表 4-9 矩阵操作函数

操作	描述
tf.diag(diagonal, name=None)	返回一个给定对角值的对角 tensor # ‘diagonal’ is [1, 2, 3, 4] tf.diag(diagonal) ==> [[1, 0, 0, 0] [0, 2, 0, 0] [0, 0, 3, 0] [0, 0, 0, 4]]
tf.diag_part(input, name=None)	功能与上面相反
tf.trace(x, name=None)	求一个 2 维 tensor 足迹，即对角值 diagonal 之和
tf.transpose(a, perm=None, name=’transpose’)	调换 tensor 的维度顺序 按照列表 perm 的维度排列调换 tensor 顺序， 如为定义，则 perm 为(n-1…0) # ‘x’ is [[1 2 3],[4 5 6]] tf.transpose(x) ==> [[1 4], [2 5],[3 6]] # Equivalently tf.transpose(x, perm=[1, 0]) ==> [[1 4],[2 5], [3 6]]

tf.reverse(a, axis)	#将轴的索引反转
	tensor=[[1, 2, 3], [4, 5, 6]]
	tf.reverse (tensor, [0, 1]) ==> [[6 5 4] [3 2 1]]

tf.matmul(a, b, transpose_a=False, transpose_b=False, a_is_sparse=False, b_is_sparse=False, name=None)	矩阵相乘
tf.matrix_determinant(input, name=None)	返回方阵的行列式
tf.matrix_inverse(input, adjoint=None, name=None)	求方阵的逆矩阵，adjoint 为 True 时，计算输入共轭矩阵的逆矩阵
tf.cholesky(input, name=None)	对输入方阵 cholesky 分解，即把一个对称正定的矩阵表示成一个下三角矩阵 L 和其转置的乘积的分解 A=LL^T
tf.matrix_solve(matrix, rhs, adjoint=None, name=None)	求解 tf.matrix_solve(matrix, rhs, adjoint=None, name=None) matrix 为方阵 shape 为[M,M],rhs 的 shape 为[M,K]，output 为[M,K]

4.2.4 复数操作函数

关于复数的操作函数见表 4-10。

表 4-10 复数操作函数

函数	描述
tf.complex(real, imag, name=None)	将两实数转换为复数形式 # tensor ‘real’ is [2.25, 3.25] # tensor imag is [4.75, 5.75]

	tf.complex(real, imag) ==> [[2.25 + 4.75j], [3.25 + 5.75j]]
tf.complex_abs(x, name=None)	计算复数的绝对值，即长度。 # tensor ‘x’ is [[-2.25 + 4.75j], [-3.25 + 5.75j]] tf.complex_abs(x) ==> [5.25594902, 6.60492229]
tf.conj(input, name=None)	计算共轭复数
tf.imag(input, name=None) tf.real(input, name=None)	提取复数的虚部和实部
tf.fft(input, name=None)	计算一维的离散傅里叶变换，输入数据类型为 complex64

4.2.5 规约计算

规约计算的操作都会有降维的功能，在所有 reduce_xxx 系列操作函数中，都是以 xxx 的手段降维。每个函数都有 axis 这个参数，即沿某个方向，使用 xxx 方法对输入的 tensor 进行降维。

axis 的默认值时 None，即把 input_tensor 降到 0 维，即一个数。

对于 2 维 input_tensor 而言： axis=0 按列计算；axis=1，按行计算。

参数 reduction_indices 是为了兼容以前版本与 axis，保证相同的含义。

表 4-11 规约计算函数

操作	描述
tf. reduce_sum(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)	计算输入 tensor 元素的和，或者安照 axis 指定的轴进行求和 # ‘x’ is [[1, 1, 1],[1, 1, 1]] tf.reduce_sum(x) ==> 6 tf.reduce_sum(x, 0) ==> [2, 2, 2] tf.reduce_sum(x, 1) ==> [3, 3] tf.reduce_sum(x, 1, keep_dims=True) ==> [[3], [3]] tf.reduce_sum(x, [0, 1]) ==> 6
tf.reduce_prod(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	计算输入 tensor 元素的乘积，或者安照 axis 指定的轴进行求乘积
tf.reduce_min(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	求 tensor 中最小值
tf.reduce_max(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	求 tensor 中最大值
tf.reduce_mean(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	求 tensor 中平均值
tf.reduce_all(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	对 tensor 中各个元素求逻辑’与’ # ‘x’ is # [[True, True] # [False, False]] tf.reduce_all(x) ==> False tf.reduce_all(x, 0) ==> [False, False]

	tf.reduce_all(x, 1) ==> [True, False]
tf.reduce_any(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	对 tensor 中各个元素求逻辑’或’
tf.accumulate_n(input_tensor, axis=None, keep_dims=False,name=None, reduction_indices=None)	计算一系列 tensor 的和 # tensor ‘a’ is [[1, 2], [3, 4]] # tensor b is [[5, 0], [0, 6]] tf.accumulate_n([a, b, a]) ==> [[7, 4], [6, 14]]
tf.cumsum(x, axis=0, exclusive=False, reverse=False, name=None)	求累积和 tf.cumsum([a, b, c]) ==> [a, a + b, a + b + c] tf.cumsum([a, b, c], exclusive=True) ==> [0, a, a + b] tf.cumsum([a, b, c], reverse=True) ==> [a + b + c, b + c, c] tf.cumsum([a, b, c], exclusive=True, reverse=True) ==> [b + c, c, 0]
tf.truncatediv (x,y)	#相除后，获取强制的基于整数截断的行为 tf.truncatediv(9,4) ==>2
tf.truncatemod (x,y)	#取余后，获取强制的基于整数截断的行为 tf.truncatediv(9,4) ==>2

4.2.6 分割

分割操作是 TensorFlow 不太常用的操作，在复杂的网络模型里偶尔才会用到。

表 4-12 分割相关函数

操作	描述
tf.segment_sum(data, segment_ids, name=None)	根据 segment_ids 的分段计算各个片段的和 其中 segment_ids 为一个 size 与 data 第一维相同的 tensor 其中 id 为 int 型数据，最大 id 不大于 size c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]]) tf.segment_sum(c, tf.constant([0, 0, 1])) ==>[[0 0 0 0] [5 6 7 8]] 上面例子分为[0,1]两 id,对相同 id 的 data 相应数据进行求和, 并放入结果的相应 id 中，且 segment_ids 只升不降
tf.segment_prod(data, segment_ids, name=None)	根据 segment_ids 的分段计算各个片段的积
tf.segment_min(data, segment_ids, name=None)	根据 segment_ids 的分段计算各个片段的最小值
tf.segment_max(data, segment_ids, name=None)	根据 segment_ids 的分段计算各个片段的最大值
tf.segment_mean(data, segment_ids, name=None)	根据 segment_ids 的分段计算各个片段的平均值
tf.unsorted_segment_sum(data, segment_ids, num_segments, name=None)	与 tf.segment_sum 函数类似， 不同在于 segment_ids 中 id 顺序可以是无序的
tf.sparse_segment_sum(data, indices, segment_ids, name=None)	输入进行稀疏分割求和 c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]]) # Select two rows, one segment.

	tf.sparse_segment_sum(c, tf.constant([0, 1]), tf.constant([0, 0])) ==> [[0 0 0 0]] 对原 data 的 indices 为[0,1]位置的进行分割， 并按照 segment_ids 的分组进行求和

4.2.7 序列比较与索引提取

对于序列和数组的操作，是本书中非常常用的方法，具体的函数见表 4-13。

表 4-13 序列比较与索引提取相关函数

操作	描述
tf.argmin(input,axis, name=None)	返回 input 最小值的索引 index
tf.argmax(input,axis, name=None)	返回 input 最大值的索引 index,axis:0 表示按列，1 表示按行，
tf. setdiff1d (x, y, name=None)	返回 x，y 中不同值的索引
tf.where(input, name=None)	返回 bool 型 tensor 中为 True 的位置 # ‘input’ tensor is #[[True, False] #[True, False]] # ‘input’ 有两个’True’,那么输出两个坐标值. # ‘input’的 rank 为 2, 所以每个坐标为具有两个维度. where(input) ==> [[0, 0], [1, 0]]
tf.unique(x, name=None)	返回一个元组 tuple(y,idx)，y 为 x 的列表的唯一化数据列表， idx 为 x 数据对应 y 元素的 index # tensor ‘x’ is [1, 1, 2, 4, 4, 4, 7, 8, 8] y, idx = unique(x) y ==> [1, 2, 4, 7, 8] idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
tf.invert_permutation(x, name=None)	置换 x 数据与索引的关系 # tensor x is [3, 4, 0, 2, 1] invert_permutation(x) ==> [2, 4, 3, 0, 1]
tf.random_shuffle（input）	#random_shuffle 沿着 value 的第一维进行随机重新排列

4.2.8 错误类

作为一个完整的框架，有它自己的错误处理。TensorFlow 中的错误类如下，该部分不是太常用，可以作为工具，使用时查一下即可。

表 4-14 错误类

操作	描述
class tf.OpError	一个基本的错误类型，在当 TF 执行失败时报错

tf.OpError.op	返回执行失败的操作节点， 有的操作如 Send 或 Recv 可能不会返回，那就要用用到 node_def 方法
tf.OpError.node_def	以 NodeDef proto 形式表示失败的 OP
tf.OpError.error_code	描述该错误的整数错误代码
tf.OpError.message	返回错误信息
class tf.errors.CancelledError	当操作或者阶段呗取消时报错
class tf.errors.UnknownError	未知错误类型
class tf.errors.InvalidArgumentError	在接收到非法参数时报错
class tf.errors.NotFoundError	当发现不存在所请求的一个实体时，比如文件或目录
class tf.errors.AlreadyExistsError	当创建的实体已经存在时报错
class tf.errors.PermissionDeniedError	没有执行权限做某操作时报错
class tf.errors.ResourceExhaustedError	资源耗尽时报错
class tf.errors.FailedPreconditionError	系统没有条件执行某个行为时报错
class tf.errors.AbortedError	操作中止时报错，常常发生在并发情形
class tf.errors.OutOfRangeError	超出范围报错
class tf.errors.UnimplementedError	某个操作没有执行时报错
class tf.errors.InternalError	当系统经历了一个内部错误时报出
class tf.errors.DataLossError	当出现不可恢复的错误 例如在运行 tf.WholeFileReader.read()读取整个文件的同时文件被删减
tf.errors.XXXXX.__init__(node_def, op, message)	使用该形式方法创建以上各种错误类

4.3 共享变量

下面来到本章的重点——共享变量。共享变量在复杂的网络中用处非常之广泛，所以读者一定要学好。

4.3.1 共享变量用途

在构建模型时，需要使用 tf.Variable 来创建一个变量（也可以理解成节点）。例如代码：

```
biases = tf.Variable(tf.zeros([2]), name=" biases")#创建一个偏执的学习参数，在训练时，这个变量不断的更新。
```

但在某种情况下，一个模型需要使用其他模型创建的变量，两个模型一起训练。比如对抗网络中的生成器模型与判别器模型（后文 12 章会有详细讲解）。如果使用 tf.Variable，将会生成一个新的变量，而我们需要的的是原来的那个 biases 变量。这时怎么办呢？

就是通过引入了 get_variable 方法,实现共享变量来解决这个问题。这个种方法可以使用多套网络模型来训练一套权重。

4.3.2 使用 get-variable 获取变量

get_variable 必须配合 variable_scope 一起使用。variable_scope 的意思就是变量作用域。在某一作用域中的变量可以被设置成共享的方式，被其他网络模型使用。

定义如下：

```
tf.get_variable(<name>, <shape>, <initializer>)
```

在 TensorFlow 里，使用 get_variable 生成的变量是以指定的 name 属性为唯一标识，并不是定义的变量名称。使用时一般通过 name 属性定位到具体变量，并将其共享到其他模型中去。

下面通过两个例子来深入介绍。

4.3.3 实例 14：演示 get_variable 和 Variable 的区别

案例描述

分别演示使用 Variable 定义变量和使用 get_variable 来定义变量。请者低仔细观察它们的用法区别。

1. Variable 的用法

首先先来看一下 Variable 的用法。

代码 4-9 get_variable 和 Variable 的区别

```
01  import tensorflow as tf
02
03  var1 = tf.Variable(1.0 , name='firstvar')
04  print ("var1:",var1.name)
05  var1 = tf.Variable(2.0 , name='firstvar')
06  print ("var1:",var1.name)
07  var2 = tf.Variable(3.0 )
08  print ("var2:",var2.name)
09  var2 = tf.Variable(4.0 )
10  print ("var1:",var2.name)
11
12  with tf.Session() as sess:
13      sess.run(tf.global_variables_initializer())
14      print("var1=",var1.eval())
15      print("var2=",var2.eval())
```

上面的代码运行后得到如下输出：

```
var1: firstvar:0
var1: firstvar_1:0
var2: Variable:0
var1: Variable_1:0
var1= 2.0
var2= 4.0
```

上面代码定义了两次 var1，可以看到在内存中生成了两个 var1（因为它们的 name 不一样），对于图来讲后面的 var1 是生效的（var1=2.0）。

var2 认明了：Variable 定义时没有指定名字，系统会自动给加上一个名字 Variable:0。

2. get_variable 用法演示

接着上面代码，使用 get_variable 添加 get_var1 变量。

代码 4-9　get_variable 和 Variable 的区别（续）

```
16  get_var1                =                tf.get_variable("firstvar",[1],
    initializer=tf.constant_initializer(0.3))
17  print ("get_var1:",get_var1.name)
18
19  get_var1                =                tf.get_variable("firstvar",[1],
    initializer=tf.constant_initializer(0.4))
20  print ("get_var1:",get_var1.name)
```

运行之后结果如下：

```
var1: firstvar:0
var1: firstvar_1:0
var2: Variable:0
var1: Variable_1:0
var1= 2.0
var2= 4.0
get_var1: firstvar_2:0
Traceback (most recent call last):
….
```

可以看到，在定义第 2 个 get_var1 发生崩溃了。这表明，使用 get_variable 只能定义一次指定名称的变量。同时由于变量“firstvar”在前面使用 Variable 函数的生成过一次，所以系统自动变成了“firstvar_2:0”。

如果将崩溃的句子改成下面的样子：

代码 4-9　get_variable 和 Variable 的区别（续）

```
21  get_var1                =                tf.get_variable("firstvar",[1],
    initializer=tf.constant_initializer(0.3))
22  print ("get_var1:",get_var1.name)
23
24  get_var1                =                tf.get_variable("firstvar1",[1],
    initializer=tf.constant_initializer(0.4))
25  print ("get_var1:",get_var1.name)
26
27  with tf.Session() as sess:
28      sess.run(tf.global_variables_initializer())
29      print("get_var1=",get_var1.eval())
```

运行代码，输出如下：（部分内容）

```
……
get_var1: firstvar_2:0
get_var1: firstvar1:0
get_var1= [ 0.40000001]
```

可以看到，这次仍然是又定义了一个 get_var1，不同的是，改变了它的名字“firstvar1”，这样是没有问题的。同样，新的 get_var1 会在图中生效，所以它的输出值是 4.0 而不是 3.0。

4.3.4　实例 15：在特定的作用域下获取变量

案例描述

在作用域下，使用 get_variable，以及嵌套 variable_scope。

在前面的例子中，已经知道使用 get_variable 创建两个同样的名字的变量是行不通的，如下代码会报错。

```
var1 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
var2 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
```

如果真的想那么做，可以使用 variable_scope 将他们隔开，如下代码：

代码 4-10　get_variable 配合 variable_scope

```
import tensorflow as tf
with tf.variable_scope("test1", ):     #定义一个作用域 test1
   var1 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)

with tf.variable_scope("test2"):
   var2 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)

print ("var1:",var1.name)
print ("var2:",var2.name)
```

运行代码，输出如下结果：

```
var1: test1/firstvar:0
var2: test2/firstvar:0
```

var1 和 var2 都使用“firstvar”的名字来定义。通过输出可以看出，其实生成的两个变量 var1 和 var2 是不同的。它们作用在不同的 scope 下。这就是 scope 的作用。

Scope 还支持嵌套，将上面代码中的第二个 scope 缩进一下，得到如下代码：

代码 4-11　get_variable 配合 variable_scope2

```
01  ……
02
03  with tf.variable_scope("test1", ):
04      var1 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
05
06      with tf.variable_scope("test2"):
07          var2 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
08
09  print ("var1:",var1.name)
10  print ("var2:",var2.name)
```

运行代码，输出如下结果：

```
var1: test1/firstvar:0
var2: test1/test2/firstvar:0
```

4.3.5　实例 16：共享变量功能的实现

案例描述

使用作用域中的 reuse 参数来实现共享变量功能。

费了这么大的劲来使用 get_variable，目的其实是为了要通过它实现共享变量的功能。

variable_scope 里面有个 reuse=True 属性，就是表明使用已经定义过的变量。这时 get_variable 将不再会创建新的变量，而是去图中 get_variable 所创建过的变量中找 name 相同的变量。

在上文代码中加入再建立一个同样的 scope，并且设置 reuse=True，实现共享“firstvar”变量。代码如下。

代码 4-11　get_variable 配合 variable_scope2（续）

```
11  with tf.variable_scope("test1",reuse=True ):
12      var3= tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
13      with tf.variable_scope("test2"):
14          var4 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
15
16  print ("var3:",var3.name)
17  print ("var4:",var4.name)
```

运行上面代码，得出如下输出：

```
var1: test1/firstvar:0
var2: test1/test2/firstvar:0
var3: test1/firstvar:0
var4: test1/test2/firstvar:0
```

var1 和 var3 的输出名字是一样的，var2 和 var4 的名字也是一样的。这表明 var1 和 var3 共用了一个变量，var2 和 var4 共用了一个变量，实现了共享变量。在实际应用中，就可以把 var1 和 var2 放到一个网络模型里去训练，把 var3 和 var4 放到另一套网络模型里去训练。而两个模型的优化结果都会作用于一套模型的学习参数上。

注意：

如果在 Anaconda 里面的 spyder 运行，该代码只能运行一次，第二次会报错。

解决办法：需要在 Anacondad 的 consoles 菜单里将当前的 kernel 退出，再重新进入一下。在运行才不会报错。否则会提示已经有这个变量了。

为什么会这样呢？

tf.get_variable 在创建变量时，会去检查图中是否已经创建过该变量。如果创建过并且本次调用时没有被设为共享方式，则会报错。

明白原理后可以加一条语句：tf.reset_default_graph()，将**图里面**的变量清空，就可以解决这个问题。图的更多内容将在后面章节介绍。

4.3.6　实例 17：初始化共享变量的作用域

案例描述

演示 variable_scope 中 get_variable 初始化的继承功能，以及嵌套 variable_scope 的继承功能。

variable_scope 和 get_variable 都有初始化的功能。在初始化时，如果没有对当前变量初始化，TensorFlow 会默认使用用域的初始化方法对其初始化。并且作用域的初始化方法也有继承功能。下面演示代码。

代码 4-12　共享变量的作用域与初始化

```
import tensorflow as tf

with tf.variable_scope("test1", initializer=tf.constant_initializer(0.4) ):
    var1 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)

    with tf.variable_scope("test2"):
        var2 = tf.get_variable("firstvar",shape=[2],dtype=tf.float32)
        var3                                                                    =
tf.get_variable("var3",shape=[2],initializer=tf.constant_initializer(0.3))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print("var1=",var1.eval())              #test1 下的
    print("var2=",var2.eval())              #test2 下的，继承 test1 初始化
    print("var3=",var3.eval())              #test2 下的
```

上述代码大致操作如下：

- 将 test1 作用域进行初始化为 4.0，见代码第 3 行；
- var1 没有初始化，见代码第 4 行；
- 嵌套的 test2 作用域也没有初始化，见代码第 6 行；
- test2 下的 var3 进行了初始化，见代码第 8 行。

运行代码，输出如下：

```
var1= [ 0.40000001  0.40000001]
var2= [ 0.40000001  0.40000001]
var3= [ 0.30000001  0.30000001]
```

var1 数组值为 0.4，表明继承了 test1 的值；var2 数组值为 0.4，表明其所在的作用域 test2 也继承了 test1 的初始化；var3 使用了自己的初始化，所以数组值为 0.3。

4.3.7 实例 18：演示作用域与操作符的受限范围

案例描述

演示 variable_scope 的 as 用法，以及对应的作用域。

variable_scope 还可以使用 with variable_scope（“name”）as xxxscope 的方式，当使用这种方式时，所定义的作用域变量“xxxscope”将不在受到外围的 scope 所限制。看下面的例子。

代码 4-13 作用域与操作符的受限范围

```
01  import tensorflow as tf
02
03  with tf.variable_scope("scope1") as sp:
04      var1 = tf.get_variable("v", [1])
05
06  print("sp:",sp.name)                        #作用域名称
07  print("var1:",var1.name)
08
09  with tf.variable_scope("scope2"):
10      var2 = tf.get_variable("v", [1])
```

```
11
12      with tf.variable_scope(sp) as sp1:
13          var3 = tf.get_variable("v3", [1])
14
15  print("sp1:",sp1.name)
16  print("var2:",var2.name)
17  print("var3:",var3.name)
```

例子中定义了作用域 scope1 as sp，然后将 sp 放在作用域 scope2 中，并 as 成 sp1。运行输出如下：

```
sp: scope1
var1: scope1/v:0
sp1: scope1
var2: scope2/v:0
var3: scope1/v3:0
```

sp 和 var1 的输出前面已经交代过。Sp1 在 scope2 下，但是输出仍是 scope1，没有改变。在它下面定义的 var3 的名字是 scope1/v3:0，表明也在 scope1 下。再次说明 sp 没有受到外层的限制。

另外再介绍一个操作符的作用域——tf.name_scope，如下所示。操作符不仅受到 tf.name_scope 作用域的限制，也同时受到 tf.variable_scope 作用域的限制。

代码 4-13　作用域与操作符的受限范围（续）

```
18  with tf.variable_scope("scope"):
19      with tf.name_scope("bar"):
20          v = tf.get_variable("v", [1])          #变量
21          x = 1.0 + v                            # x 为 op
22  print("v:",v.name)
23  print("x.op:",x.op.name)
```

上面的代码运行后输出：

```
v: scope/v:0
x.op: scope/bar/add
```

可以看到，虽然 v 和 x 都在 scope 的 bar 下面，但是 v 的命名只受到 scope 的限制，tf.name_scope 只能限制住 op，不能限制变量的命名。

在 tf.name_scope 函数中，还可以使用空字符将作用域返回到顶层。

下面举例来比较 tf.name_scope 与 variable_scope 在空字符情况下的处理：

- 在代码第 28 行 var3 的定义之后添加空字符的 variable_scope，
- 定义 var4，见代码第 31 行。

代码 4-13　作用域与操作符的受限范围（续）

```
24  with tf.variable_scope("scope2"):
25      var2 = tf.get_variable("v", [1])
26
27      with tf.variable_scope(sp) as sp1:
28          var3 = tf.get_variable("v3", [1])
29
```

```
30          with tf.variable_scope("") :
31              var4 = tf.get_variable("v4", [1])
```

在 x = 1.0 + v 之后添加空字符的 tf.name_scope，并定义 y。如下代码：

代码 4-13　作用域与操作符的受限范围（续）

```
32  with tf.variable_scope("scope"):
33      with tf.name_scope("bar"):
34          v = tf.get_variable("v", [1])
35          x = 1.0 + v
36          with tf.name_scope(""):
37              y = 1.0 + v
```

将 var4 和 y 的值打印出来，得出如下信息:

```
var4: scope1//v4:0
y.op: add
```

可以看到，y 变成顶层了，而 var4 多了一个空层。

4.4　实例 19：图的基本操作

前面接触了一些图的概念，这里来系统的了解一下 TensorFlow 中的图可以做哪些事情。

案例描述

(1) 演示使用三种方式来建立图，并依次设置为默认图，使用 get_default_graph 方法来获取当前默认图，验证默认图的设置生效。

(2) 演示获取图中相关内容的操作。

一个 TensorFlow 程序是默认建立一个图的，除了系统自动建图以外，还可以手动建立，并做一些其他的操作。

4.4.1　建立图

可以在一个 TensorFlow 中手动建立其他的图，也可以根据图里面的变量获得当前的图。

下面代码中演示了：使用 tf.Graph()建立图，使用 tf.get_default_graph()获得图，还有使用 reset_default_graph()来重置图。

代码 4-14　图的基本操作

```
01  import numpy as np
02  import tensorflow as tf
03  c = tf.constant(0.0)
04
05  g = tf.Graph()
06  with g.as_default():
07    c1 = tf.constant(0.0)
08    print(c1.graph)
09    print(g)
10    print(c.graph)
11
12  g2 =  tf.get_default_graph()
13  print(g2)
14
```

```
15  tf.reset_default_graph()
16  g3 =  tf.get_default_graph()
17  print(g3)
```

上面代码运行结果如下显示：

```
<tensorflow.python.framework.ops.Graph object at 0x000000000B854940>
<tensorflow.python.framework.ops.Graph object at 0x000000000B854940>
<tensorflow.python.framework.ops.Graph object at 0x000000000923CCF8>
<tensorflow.python.framework.ops.Graph object at 0x000000000923CCF8>
<tensorflow.python.framework.ops.Graph object at 0x000000000B8546D8>
```

可以看出，

（1）c 是在刚开始的默认图中建立的，所以它的图的打印值就是原始的默认图的打印值 923CCF8,

（2）后来使用 tf.Graph()建立了一个图 B854940，并且在新建的图里添加变量，可以通过变量的“.graph”获得所在的图。

（3）在新图 B854940 的作用域外，使用 tf.get_default_graph()又获得了原始的默认图 923CCF8。接着又使用了 tf.reset_default_graph()函数，它相当于从新建了一张图来代替原来的默认图，这时默认的图变成了 B8546D8。

4.4.2 获取张量

在图里面可以通过名字得到其对应的元素，例如，get_tensor_by_name 可以获得图里面的张量。

在上个实例中添加代码如下。

代码 4-14 图的基本操作（续）

```
18  print(c1.name)
19  t = g.get_tensor_by_name(name = "Const:0")
20  print(t)
```

该部分代码运行结果如下显示：

```
Const:0
Tensor("Const:0", shape=(), dtype=float32)
```

常量 C1 是在一个子图 g 中建立的。with tf.Graph() as default 代码表示使用 tf.Graph()来创建一个图，并在其上面定义 OP，见代码第 5、6 行。

接着演示了如何访问该图中的变量：将 c1 的名字放到 get_tensor_by_name 里面来反向得到其张量（见代码第 20 行），通过对 t 的打印可以看到所得的 t 就是前面定义的张量 c1。

注意：

不必太花精力去关注 TensorFlow 中默认的命名规则。一般在需要使用名字时，都会在定义的同时为它指定好固定的名字。如果真要不清楚某个元素的名字，将其打印出来，回填到代码中，再次运行即可。

4.4.3 获取操作符

获取操作符（Operation，OP）的方法和获取张量的方法非常类似，使用的方法是

get_operation_by_name（）。下面将获取张量和获取 OP 的例子放在一起比较一下，具体见如下代码。

代码 4-14　图的基本操作（续）

```
a = tf.constant([[1.0, 2.0]])
b = tf.constant([[1.0], [3.0]])

tensor1 = tf.matmul(a, b, name='exampleop')
print(tensor1.name,tensor1)
test = g3.get_tensor_by_name("exampleop:0")
print(test)

print(tensor1.op.name)
testop = g3.get_operation_by_name("exampleop")
print(testop)

with tf.Session() as sess:
    test =  sess.run(test)
    print(test)
    test = tf.get_default_graph().get_tensor_by_name("exampleop:0")
    print (test)
```

上面示例中，先将张量及其名字打印出来，然后使用 g3 图的 get_tensor_by_name 函数又获得到了该张量，此时 test 和 tensor1 是一样的。为了证明这一点，直接把 test 放到 session 的 run 里，发现它也能运行出正确的结果。

注意：

使用默认的图时，也可以用上述代码中的 tf.get_default_graph()获取当前图，然后接着可以调用 get_tensor_by_name 获取元素。

上面代码运行后会显示如下信息：

```
exampleop:0 Tensor("exampleop:0", shape=(1, 1), dtype=float32)
Tensor("exampleop:0", shape=(1, 1), dtype=float32)
exampleop
name: "exampleop"
op: "MatMul"
input: "Const"
input: "Const_1"
attr {
  key: "T"
  value {
    type: DT_FLOAT
  }
}
attr {
  key: "transpose_a"
```

```
    value {
      b: false
    }
  }
  attr {
    key: "transpose_b"
    value {
      b: false
    }
  }
  [[ 7.]]
  Tensor("exampleop:0", shape=(1, 1), dtype=float32)
```

再仔细看上例中的 op，通过打印 tensor1.op.name 的信息，获得了 op 的名字，然后通过 get_operation_by_name 获得了相同的 op，可以看出 op 与 tensor1 之间的对应关系。

注意：

这里之所以要放在一起举例，原因就是 op 和张量在定义节点时很容易被混淆。上例子中的 tensor1 = tf.matmul(a, b, name='exampleop')并不是 op，而是张量。Op 其实是描述张量中的运算关系，是通过访问张量的属性可以找到的。

4.4.4 获取元素列表

如果想看一下图中的全部元素，可以使用 get_operations()来实现。具体代码如下。

代码 4-14 图的基本操作（续）

```
38  tt2 = g.get_operations()
39  print(tt2)
```

运行后显示如下信息：

```
[<tf.Operation 'Const' type=Const>]
```

由于 g 里面只有一个常量，所以打印了一条信息。

4.4.5 获取对象

前面是多根据名字来获取元素，还可以根据对象来获取元素。使用 tf.Graph.as_graph_element(obj, allow_tensor=True, allow_operation=True)函数，即，传入的是一个对象，返回一个张量或是一个 op。该函数具有验证和转换功能，在多线程方面会偶尔用到。举例如下。

代码 4-14 图的基本操作（续）

```
40  tt3 = g.as_graph_element(c1)
41  print(tt3)
```

运行输出结果如下：

```
Tensor("Const:0", shape=(), dtype=float32)
```

上述的代码是将自己传进来，返回的仍然是自己，只是名字由 c1 变成了 tt3。

备注：

这里只是介绍了图中比较简单的操作，图的操作还有很多，有的还很常用。但考虑到初学者的接受程度，更复杂的图操作（比如冻结图，一个图导入到另一个图等）将会在后面再来介绍。

4.4.6 练习题

试试将 tf.get_default_graph()放在 with tf.Graph().as_default():作用域里，看看会得到什么。是全局的默认图，还是 tf.Graph()新建的图？（示例代码在 4-14 图的基本操作中）

4.5 配置分布式 TensorFlow

在大型的数据集上进行神经网络的训练，往往需要更大的运算资源，而且还要花上若干天才能完成运算量。

TensorFlow 提供了一个可以分布部署的模式，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，这样使用计算机群运算来代替单机计算，可以时训练时间大大变短。

4.5.1 分布式 TensorFlow 的角色及原理

要想配置 TensorFlow 为分布训练，需要先了解 TensorFlow 中关于分布式的角色分配。

- ps：作为分布式训练的服务端，等待各个终端（supervisors）来连接；
- worker:在 TensorFlow 的代码注释中被称作 supervisors,作为分布式训练的运算终端；
- chief supervisors：在众多运算终端中必须选择一个作为主要的运算终端。该终端是在运算终端中最先启动的，它的功能是合并各个终端运算后的学习参数，将其保存或载入。

每个具体的角色的网络标识都是唯一的，即，分布在不同 IP 的机器上（或者同一个机器，但是不同端口）。

在实际运行中,各个角色的网络构建部分代码必须 100%的相同。三者的分工如下：

- 服务端作为一个多方协调者，等待各个运算终端来连接；
- chief supervisors 会在启动时统一管理全局的学习参数，进行初始化或从模型载入。
- 其他的运算终端只是负责得到其对应的任务并进行计算，并不会保存检查点，用于 TensorBoard 可视化中的 summary 日志等任何参数信息。

整个过程都是通过 RPC 协议来通信的。

4.5.2 分布部署 TensorFlow 的具体方法

配置过程中，首先需要建一个 server，在 server 中会将 ps 及所有 worker 的 IP 端口准备好。接着，使用 tf.train.Supervisor 中的 managed_session 来管理一个打开的 session。Session 中只是负责运算，而通信协调的事情就都交给 Supervisor 来管理了。

4.5.3 实例 20：使用 TensorFlow 实现分布式部署训练

下面就开始实现一个分布式训练的网络模型。本例以“4-8 线性回归的 TensorBoard 可视化.py”为原型，在其中添加代码将其改成分布式。

案例描述

在本机通过 3 个端口来建立 3 个终端，分别为一个 ps，两个 worker，实现 TensorFlow 的分布式运算。

具体步骤如下：

1．为每个角色添加 IP 地址和端口，创建 server

在一台机器上开 3 个不同的端口，分别代表 ps、chief supervisors 、worker。角色的名称用 strjob_name 表示。以 ps 为例，代码如下：

代码 4-15　ps

```
……
#定义 ip 和端口
strps_hosts="localhost:1681"
strworker_hosts="localhost:1682,localhost:1683"

#定义角色名称
strjob_name = "ps"
task_index = 0
#将字符串转成数组
ps_hosts = strps_hosts.split(',')
worker_hosts = strworker_hosts.split(',')
cluster_spec = tf.train.ClusterSpec({'ps': ps_hosts,'worker': worker_hosts})
#创建 server
server = tf.train.Server(
                 {'ps': ps_hosts,'worker': worker_hosts},
                 job_name=strjob_name,
                 task_index=task_index)
```

注意：

没有网络基础的读者可能看不明白 localhost，说好的 IP 地址呢？localhost 及是本机域名的写法，等同于 127.0.0.1（本机 ip）。如果是跨机器来做分布式训练，直接写成对应机器的 ip 地址即可。

2. 为 ps 角色添加等待函数

ps 角色使用 server.join()函数进行线程挂起，开始接收连接消息。

代码 4-15　ps（续）

```
#ps 角色使用 join 进行等待
if strjob_name == 'ps':
  print("wait")
  server.join()
```

3 .创建网络结构

与正常的程序不同，在创建网络结构时，使用 tf.device 函数将全部的节点都放在当前任务下。

在 tf.device 中的任务是通过 tf.train.replica_device_setter 来指定的。

在 tf.train.replica_device_setter 中使用 worker_device 来定义具体任务名称；使用 cluster 的配置来指定角色及对应的 IP 地址，从而实现管理整个任务下的图节点。

代码 4-15　ps（续）

```
22  with tf.device(tf.train.replica_device_setter(
23              worker_device="/job:worker/task:%d" % task_index,
24              cluster=cluster_spec)):
25      X = tf.placeholder("float")
26      Y = tf.placeholder("float")
27      # 模型参数
28      W = tf.Variable(tf.random_normal([1]), name="weight")
29      b = tf.Variable(tf.zeros([1]), name="bias")
30
31      global_step = tf.contrib.framework.get_or_create_global_step()#获得迭代
    次数
32
33      # 前向结构
34      z = tf.multiply(X, W)+ b
35      tf.summary.histogram('z',z)                          #将预测值以直方图显示
36      #反向优化
37      cost =tf.reduce_mean( tf.square(Y - z))
38      tf.summary.scalar('loss_function', cost)          #将损失以标量显示
39      learning_rate = 0.01
40      optimizer                                                                  =
    tf.train.GradientDescentOptimizer(learning_rate).minimize(cost,global_ste
    p=global_step) #梯度下降
41
42      saver = tf.train.Saver(max_to_keep=1)
43      merged_summary_op = tf.summary.merge_all()        #合并所有 summary
44
45      init = tf.global_variables_initializer()
```

为了让载入检查点文件时能够同步循环次数，这里加了一个 global_step 变量，并将其放到优化器中。这样，每次运行一次优化器，global_step 就会自动获得当期迭代的次数。

注意：

init = tf.global_variables_initializer()函数是将其前面的变量全部初始化，如果后面再有变量，则不会被初始化。所以，一般要将 init = tf.global_variables_initializer()放在最后。这是个很容易出错的地方，常常令开发者找不到头绪。可以试着在最前面运行，看看会发生什么。

4. 创建 Supervisor，管理 session

在 tf.train.Supervisor 中，is_chief 表明了是否为 chief supervisors 角色。这里将 task_index=0 的 worker 设置成 chief supervisors。

Logdir 为检查点文件和 summary 文件保存的路径。

init_op 表示使用初始化变量的函数。

saver 需要将保存检查点的 saver 对象传入，Supervisor 就会自动保存检查点文件。如果不想自动保存，可以设为 None。

同理，summary_op 也是自动保存 summary 文件。这里设为 None，表示不自动保存。

save_model_secs 为保存检查点文件的时间间隔。这里设为 5，表示每 5 秒自动保存一次检查点文件。

代码 4-15　ps（续）

```
46  # 定义参数
47  training_epochs = 2200
48  display_step = 2
49
50  sv = tf.train.Supervisor(is_chief=(task_index == 0),#0 号 worker 为 chief
51                           logdir="log/super/",
52                           init_op=init,
53                           summary_op=None,
54                           saver=saver,
55                           global_step=global_step,
56                           save_model_secs=5)
57
58  #连接目标角色创建 session
59  with sv.managed_session(server.target) as sess:
```

如上代码，为了让分布运算的效果明显一些，将迭代次数改成了 2200，使其运算时间变长。

5. 迭代训练

Session 中的内容与以前一样，直接迭代训练即可。由于使用了 Supervisor 管理 session，将使用 sv.summary_computed 函数来保存 summary 文件。同样，如想要手动保存检测点文件，也可以使用 sv.saver.save。代码如下：

代码 4-15　ps（续）

```
60  print("sess ok")
61      print(global_step.eval(session=sess))
62
63      for                                  epoch                                  in
    range(global_step.eval(session=sess),training_epochs*len(train_X)):
64
65          for (x, y) in zip(train_X, train_Y):
66              _, epoch = sess.run([optimizer,global_step] ,feed_dict={X: x, Y:
    y})
67              #生成 summary
68              summary_str = sess.run(merged_summary_op,feed_dict={X: x, Y: y});
69              #将 summary 写入文件
70              sv.summary_computed(sess, summary_str,global_step=epoch)
71              if epoch % display_step == 0:
72                  loss = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
73                  print  ("Epoch:",  epoch+1,  "cost=",  loss,"W=",  sess.run(W),
    "b=", sess.run(b))
74                  if not (loss == "NA" ):
75                      plotdata["batchsize"].append(epoch)
76                      plotdata["loss"].append(loss)
77
78      print (" Finished!")
79
    sv.saver.save(sess,"log/mnist_with_summaries/"+"sv.cpk",global_step=epoch
    )
80
81  sv.stop()
```

注意：

（1）在设置自动保存检查点文件后，手动保存仍然有效。

（2）在运行一半后终止，再运行 Supervisor 时会自动载入模型的参数，不需要再手动调用 saver.restore。

（3）在 session 中，不再需要运行 tf.global_variables_initializer()。原因是，Supervisor 在建立时会调用传入的 init_op 进行初始化，如果加了 sess.run(tf.global_variables_initializer())，会导致所载入模型的变量被二次清空。

6. 建立 worker 文件

将文件复制 2 份，分别起名为“4-16 worker.py”与“4-17　worker2.py”，将角色名称修改成 worker，并将“4-16 worker2.py”中的 task_index 设为 1。

代码 4-16　worker

```
……
#定义角色名称
strjob_name = "worker"
task_index = 0
……
```

代码 4-17　worker2

```
……
#定义角色名称
strjob_name = "worker"
task_index = 1
……
```

注意：

这里不能使用 sv.summary_computed，因为 worker2 不是 chief supervisors，在 worker2 中是不会为 Supervisor 对象构造默认 summary_writer 的，所以即使调用 summary_computed 也无法执行下去，程序会报错。

手写控制 summary 与检查点文件保存时，需要将 chief supervisors 以外的 worker 全部去掉才可以。要么就使用 Supervisor 按时间间隔保存的形式来管理，这样就可以一套代码解决了。

7. 部署运行

（1）在 Spyder 中先将“4-15ps.py”运行起来，单击菜单“Consoles/Open an IPython console”来新打开一个 consoles，如图 4-7 所示。

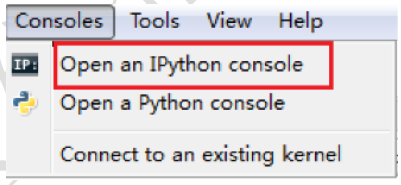

图 4-7 consoles 菜单

（2）在面板的右下角出现一个 consoles 框（consoles2/A），单点击新建的 consoles 窗口，确保为激活状态，如图 4-8 所示。

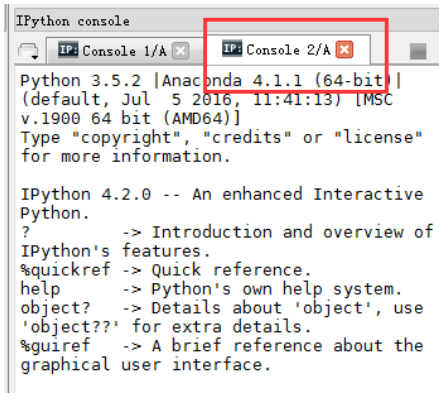

图 4-8 consoles 框

（3）运行“4-17worker2.py”文件。最后按照“4-17worker2.py”启动的方式，再将“4-16worker.py”启动，这时 3 个窗口显示内容分别如下：

- “4-16worker.py”对应窗口会显示正常的训练信息。

……

Epoch: 8000 cost= 0.0754263 W= [2.01029539] b= [-0.00388618]

Epoch: 8002 cost= 0.074845 W= [2.00651097] b= [0.00453186]

Epoch: 8003 cost= 0.0748089 W= [2.00529122] b= [0.00281144]

Epoch: 8005 cost= 0.0747555 W= [2.00324082] b= [0.00635108]

Epoch: 8007 cost= 0.075026 W= [2.00662613] b= [-0.00956773]

Epoch: 8009 cost= 0.0749311 W= [2.00585985] b= [-0.006533]

Epoch: 8010 cost= 0.0748186 W= [2.00469637] b= [-0.00152527]

Epoch: 8011 cost= 0.0750369 W= [2.0065136] b= [-0.02676161]

Epoch: 8012 cost= 0.0758979 W= [2.0068512] b= [-0.02852018]

Epoch: 8013 cost= 0.0759059 W= [2.00671506] b= [-0.02870713]

Epoch: 8015 cost= 0.0753608 W= [2.0055182] b= [-0.01959283]

Epoch: 8018 cost= 0.0760464 W= [2.00559783] b= [-0.03230772]

Epoch: 8021 cost= 0.0758819 W= [2.00522184] b= [-0.02836083]

Epoch: 8023 cost= 0.0758949 W= [2.00778055] b= [-0.01191433]

Epoch: 8026 cost= 0.0752242 W= [2.00646138] b= [-0.01574964]

Epoch: 8028 cost= 0.0751021 W= [2.00708318] b= [-0.01172168]

Epoch: 8030 cost= 0.0749788 W= [2.0083425] b= [-0.00503741]

Epoch: 8034 cost= 0.0750521 W= [2.00837708] b= [-0.0084667]

Epoch: 8035 cost= 0.0750075 W= [2.01157689] b= [0.00467709]

Epoch: 8037 cost= 0.0751661 W= [2.01191807] b= [0.0159377]

Epoch: 8038 cost= 0.0750556 W= [2.01164842] b= [0.01059892]

Epoch: 8040 cost= 0.0753085 W= [2.01313496] b= [0.01954099]

Epoch: 8042 cost= 0.0753466 W= [2.01260543] b= [0.02123925]

……

可以看到循环的次数并不是连续的，跳过的步骤被分配到 worker2 中去运算了。

- “4-17worker2.py”窗口显示的信息。

```
INFO:tensorflow:Waiting for model to be ready.  Ready_for_local_init_op:  None, ready: Variables not initialized: weight, bias, global_step
INFO:tensorflow:Starting queue runners.
……
Epoch: 8003 cost= 0.0977818 W= [ 2.00529122] b= [ 0.00281144]
Epoch: 8005 cost= 0.0979236 W= [ 2.00324082] b= [ 0.00635108]
Epoch: 8007 cost= 0.0978101 W= [ 2.0065136] b= [-0.01009204]
Epoch: 8012 cost= 0.0985371 W= [ 2.00671506] b= [-0.02870713]
Epoch: 8015 cost= 0.0981559 W= [ 2.0055182] b= [-0.01959283]
Epoch: 8017 cost= 0.0986897 W= [ 2.00519013] b= [-0.02992464]
Epoch: 8018 cost= 0.0987787 W= [ 2.00559783] b= [-0.03128441]
Epoch: 8020 cost= 0.0988223 W= [ 2.00550485] b= [-0.02906012]
Epoch: 8022 cost= 0.0985962 W= [ 2.00522184] b= [-0.02918861]
Epoch: 8024 cost= 0.0982481 W= [ 2.00616717] b= [-0.02256276]
Epoch: 8025 cost= 0.0977918 W= [ 2.00778055] b= [-0.01191433]
Epoch: 8026 cost= 0.0979684 W= [ 2.00646138] b= [-0.01574964]
Epoch: 8028 cost= 0.0978234 W= [ 2.00708318] b= [-0.01172168]
Epoch: 8030 cost= 0.0976372 W= [ 2.00842071] b= [-0.00485691]
Epoch: 8031 cost= 0.0976208 W= [ 2.00859952] b= [-0.00408681]
Epoch: 8032 cost= 0.0976431 W= [ 2.0083425] b= [-0.00503741]
Epoch: 8034 cost= 0.097557 W= [ 2.01164842] b= [ 0.01059892]
Epoch: 8039 cost= 0.0975473 W= [ 2.01065278] b= [ 0.00720035]
Epoch: 8040 cost= 0.0977502 W= [ 2.01313496] b= [ 0.01954099]
Epoch: 8042 cost= 0.0978443 W= [ 2.01260543] b= [ 0.02123925]
……
```

显示结果中有警告输出，这是因为在构建 Supervisor 时没有填写 local_init_op 参数，该参数的意思是在创建 worker 实例时，初始化本地变量。由于例子中没有填，系统就会自动初始化，并给个警告提示。

从日志中可以看到 worker2 与 chief supervisors 的迭代序号近似互补，为什么没有绝对互补呢？

分析可能是与 Supervisor 中的同步算法有关。

分布运算目的是为了提高整体运算速度，如果同步 epoch 的准确度需要以牺牲总体运算速度为代价，自然很不合适。所以更合理的推断是因为单机单次的运算太快迫使算法使用了更宽松的同步机制。

重要的一点是对于指定步数的学习参数 b 和 w 是一致的(比如第 8040 步,学习参数是相同的，都为 W= [2.01313496] b= [0.01954099])，这表明两个终端是在相同的起点上进行运算的。

- 对于“4-15ps.py”窗口则是一直静默着只显示打印的那句 wait，因为它只负责连接不

参与运算。

代码医生

qq 群：40016981