第 3 章

TensorFlow 基础

本章将向读者介绍 TensorFlow 基础，包括 TensorFlow 的基本框架、TensorFlow 模型相关内容及 TensorBoard 的基本使用方法。

值得注意的是，本书采用的 TensorFlow 版本为 1.14.0，此时 TensorFlow 框架已经在向 TensorFlow 2 转型，因此许多 TensorFlow 1 版本的 API 都同时存在于 tf 模块（TensorFlow 2 中不可用）与 tf.compat.v1 模块（TensorFlow 2 中可用）中。为了提高程序的兼容性，本书使用 tf.compat.v1 模块中的 API 进行说明（TensorFlow 1.14.0 之前的版本没有 tf.compat.v1 模块，此时读者直接使用 tf 模块中的相应方法即可）。

3.1 TensorFlow 的基本原理

想要了解 TensorFlow，需要先从 TensorFlow 的名字理解其大致原理。

TensorFlow 可以拆解为两个词进行理解，即 Tensor 与 Flow。其中 Tensor 是贯穿整个深度学习的重要概念，也是 TensorFlow 的设计灵魂，其中文意义是张量，读者可以将其简单理解为高维矩阵，或与 NumPy 中的 ndarray 进行类比。如图 3-1 所示，无形状的数据称作标量，一维数据称作矢量，二维数据称作矩阵，更高维度的数据我们就可以称其为张量。明白这一点后，我们就可以对张量的属性进行一些说明，我们称张量的维数为它的阶，它的

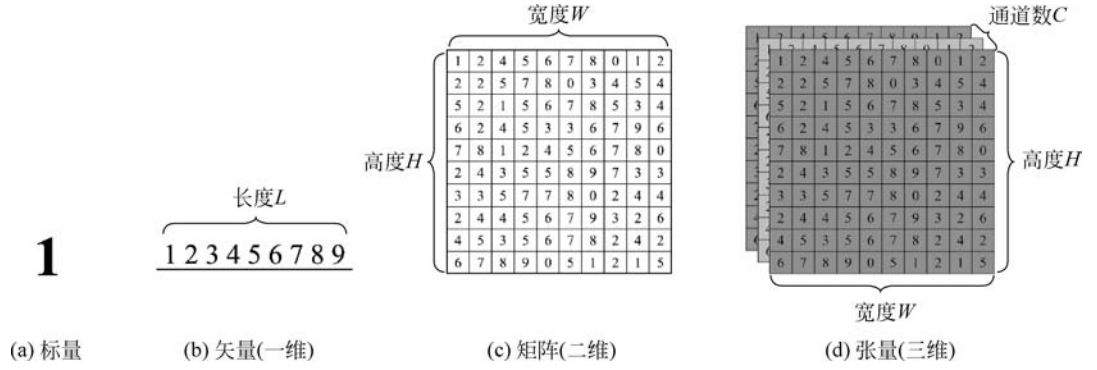

图 3-1 标量、矢量、矩阵及张量的联系与区别

形状是一个整数元组，其指定了阵列每个维度的长度。

通过观察图 3-1 可以发现，长度为 L 的一维矢量是将 L 个标量堆叠的结果；高度为 H、宽度为 W 的二维矩阵是将 H 个(W 个)长度为 W(H)的一维矢量堆叠的结果；高度为 H、宽度为 W、通道数为 C 的三维张量是将 C 个高度为 H、宽度为 W 的二维矩阵堆叠的结果。以此类推，k 维的张量可以通过堆叠 N 个 $k-1$ 维张量得到。

例如一张图像形状为(128, 128, 3)(高度为 128px，宽度为 128px，3 通道图像)，则可以说这张图像就是一个张量，它的维数(阶)为 3，其形状为(128, 128, 3)。如果有 10 张形状为(128, 128, 3)的图像该如何表示呢？我们将 10 张图像(三维张量)堆叠(stack)在一起(参照 NumPy 中的 stack 方法)，得到形状为(10,128,128,3)的张量，其中第一维表示图像数量。此时该张量为四维张量，其形状可以抽象为(N,H,W,C)，其中 N 表示张量中的图像数量(三维张量的数量)，H、W、C 分别表示图像的高度、宽度和通道数。这种四维张量表示数据的形式也是 TensorFlow 中模型图像输入采用的常用形式。

Flow 则表示张量 Tensor 在模型中“流动”的过程。具体而言，是指输入的图像(或其他数据)张量 Tensor 从模型输入层“流动”到模型输出层，变为输出张量 Tensor。在“流动”过程中，张量会不断发生改变，从图像张量(或其他输入数据)变换为输出张量(预测结果)，如图 3-2 所示。

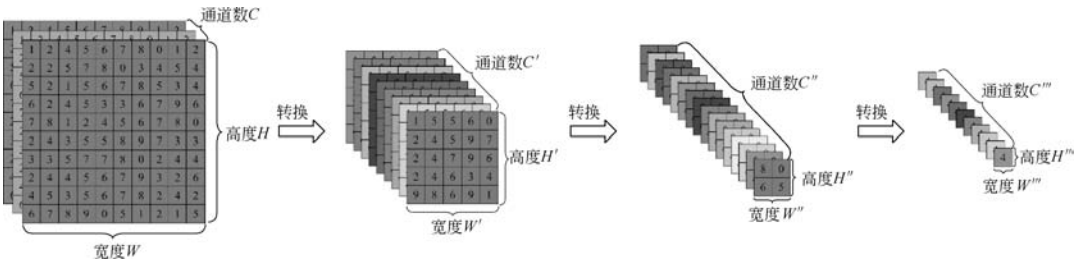

图 3-2 张量流动/转换过程

分别理解了 Tensor 与 Flow，则能明白 TensorFlow 是通过不断将输入张量进行转换得到最终预测的输出张量，以从训练数据中学习到转换规则，从而完成对数据的学习(本质上是一个拟合过程)。最终达到给定输入数据，模型给出其对应预测的过程，显示出其“智能”。

3.2 TensorFlow 中的计算图与会话机制

TensorFlow(仅限 TensorFlow 1.x 中，TensorFlow 2.x 已将会话机制删除)的核心概念就是它的计算图及其会话机制。

计算图定义了从数据输入到处理，最终到模型输出的依赖关系，TensorFlow 将每个操作(数据输入及数值计算)都抽象为一个节点，通过节点之间的依赖关系画出整张计算图。若需要得到计算图中某一个节点的值(或执行某一节点的运算)，则需要通过会话查看当前

节点的值(执行当前节点的运行)。由于节点间存在依赖关系,当尝试查看某节点的值(执行某节点的运算)时,该节点依赖的所有前驱节点都会被计算(执行)。

下面分别详细介绍计算图与会话机制的基本概念。

3.2.1 计算图

图(Graph)由顶点的有穷非空集合和顶点之间边的集合组成,通常以 $G(V,E)$ 表示。其中,G 表示一张图,V 是图 G 中顶点的集合,E 是图 G 中边的集合。类似地,TensorFlow 的计算图也有顶点 V 与边 E 的概念。其中顶点 V 即图中的计算节点,可以是 TensorFlow 中的任何操作,如张量的相加、数据的输入等,而图中的边则表示张量数据,数据在不同的操作节点之间传递("流动"),以完成数据的处理与学习。

如图 3-3 所示,表示一个最多可以完成算术运算 $(X+Y)\times Z$ 的计算图。需要注意的是,这里所表述的"最多"是指该计算图除了能完成 $(X+Y)\times Z$ 以外,它也可以用于完成 $X+Y$ 运算,下面再来详细解读这一计算图。

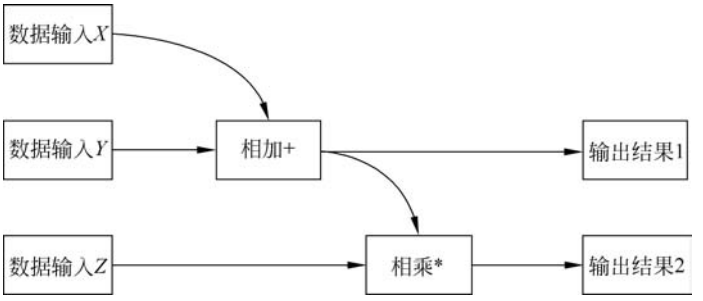

图 3-3 $(X+Y)\times Z$ 的计算图

首先,图中有 3 个数据输入节点,分别用于接收输入的 X、Y、Z,以及两个数据操作节点"相加+"和"相乘*"分别完成两个数的加与乘操作,还有两个输出节点(可选)。

当尝试得到结果 2 时,可以得知"输出结果 2"依赖于前驱节点"相乘*","相乘*"节点依赖于"数据输入 Z"和"相加+"节点,"相加+"节点依赖于"数据输入 X"与"数据输入 Y"节点。至此,所有节点依赖关系都已经探寻完毕,因此需要得到结果 2,就需要输入数据 X、Y 及 Z。同理,若只需得到结果 1,通过节点间的依赖关系可以得知,此时仅需要输入数据 X 和 Y,而不需要 Z。

可以发现,使用计算图表达运算后,整个过程十分清晰,运算之间的依赖关系也一目了然。除此之外,计算图还有一大好处就是它可以定义一个抽象的运算过程,不依赖于具体的值。如图 3-3 所示的计算图仅仅表达该图是一个最多可以完成 $(X+Y)\times Z$ 运算的图,而没有具体输出的结果值,具体的输出结果值只依赖于输出节点(输出结果 1 或输出结果 2)与具体的输入参数值(输入的 X、Y 与 Z)。

下面的程序说明了如何使用 TensorFlow 定义 $(X+Y)\times Z$ 的计算图,其中 tf.placeholder 函数将在 3.3 节中具体说明,在此读者只要从字面上理解该函数是为变量创建

了占位符(占据一个位置,而不提供具体值)即可,代码如下:

```
//ch3/define_graph.py
import tensorflow.compat.v1 as tf

#为输入变量创建占位符,并为每个变量命名
X = tf.placeholder(dtype=tf.float32, name='X')
Y = tf.placeholder(dtype=tf.float32, name='Y')
Z = tf.placeholder(dtype=tf.float32, name='Z')

#结果1为X与Y相加
result1 = X + Y
#结果2为(X + Y) * Z
result2 = result1 * Z
```

运行以上程序,则能建立如图 3-3 的计算图。

3.2.2 会话机制

在 TensorFlow 中,使用会话来运行计算图。会话中存在 fetch(取回)与 feed(注入)操作,其中 fetch 操作表示用户期望运行的操作节点,而 feed 操作则是指为计算图注入数据。

fetch 操作需要用户为会话传入需要运行的计算图中阶段,如果一次想要运行多个节点,可以以列表的形式传入 fetch 以同时得到多个结果,需要注意的是,会话运行得到的结果结构、形状与传入的 fetch 张量相同。

feed 操作则是指为计算图注入数据,这里的注入数据可以分为两种情况,一种是如 3.2.1 节所使用的 placeholder 函数,由于 placeholder 不提供具体值,因此在使用会话运行计算图时,必须为依赖输入数据的节点注入数据。第二种情况,也可以使用 feed 操作临时改变计算图中的节点值,例如在计算图中定义了一个值为 5 的节点 a,在运行计算图时,可以临时将节点 a 的值改变为自己想要的值,这种 feed 仅改变这一次运算中的 a 节点值,不影响之后的运行。注入数据使用字典的形式,即{变量 1:值 1,变量 2:值 2,…}。

TensorFlow 提供了两种会话,分别是 tf.Session()和 tf.InteractiveSession()。使用会话前需要先定义会话变量 Session,再使用 Session 对应的运行节点方法得到结果,这也是这两种会话主要不同的地方,下面就分别对这两种会话加以说明。

tf.Session()适用于运行已经将所有节点定义好的计算图,适用于在 Python 脚本中使用。第一步先使用 tf.Session 函数定义会话变量 sess,再使用 sess.run 函数运行图中的节点(fetch)与注入数据(feed)即可。如图 3-4 所示具体说明了 fetch 与 feed 在 run 函数中的用法,第一个参数可以传入单个节点或使用任意嵌套的列表结构传入多个节点,第二个参数根据待运行的节点依赖关系或用户意愿以字典的形式传入待注入数据。

在使用完成后,为了节约计算机运行资源,需要关闭会话。tf.Session()使用方法的代码如下:

图 3-4　使用 tf.Session 创建的会话运行节点的方法

```
//ch3/session.py
import tensorflow.compat.v1 as tensorflow
#导入 3.2.1 节所定义的计算图
from define_graph import *

#方法 1:手动开启/关闭会话
#sess = tf.Session()
#... 运行计算图
#关闭会话
#sess.close()

#方法 2:使用 with 语句让程序自动管理变量(推荐)
with tf.Session() as sess:
    #运行 result1(仅依赖 X 与 Y 变量),为 X 和 Y 变量分别赋值 1 和 2
    r1 = sess.run(result1, feed_dict={X: 1, Y: 2})

    #运行 result2(依赖 X、Y 与 Z 变量),为 X、Y 和 Z 变量分别赋值 1、2 和 3
    r2 = sess.run(result2, feed_dict={X: 1, Y: 2, Z: 3})

    #运行 result1 和 result2,为 X、Y、Z 变量分别赋值 4、5、6
    r3 = sess.run([result1, result2], feed_dict={X: 4, Y: 5, Z: 6})

    #打印不同的运行结果
    print(r1, r2, r3)

    #运行 result1 和 result2,为 X、Y 变量分别赋值 7、8
    r4 = sess.run([result1, result2], feed_dict={X: 7, Y: 8})
```

运行以上程序,可以得到如图 3-5 所示的结果。从结果中可以看到,r1、r2 及 r3 的结果分别为 3.0、9.0 和[9.0，54.0](由于传入的张量为一个列表,因此结果也是列表),可以看出,当为 feed_dict 传入不同值时,其具体结果也会不一样,这也进一步说明了该计算图定义

```
3.0 9.0 [9.0, 54.0]
Traceback (most recent call last):
  File "D:\Software\Anaconda3\envs\tf_gpu\lib\site-packages\tensorflow\python\client\session.py", line 1356, in _do_call
    return fn(*args)
  File "D:\Software\Anaconda3\envs\tf_gpu\lib\site-packages\tensorflow\python\client\session.py", line 1341, in _run_fn
    options, feed_dict, fetch_list, target_list, run_metadata)
  File "D:\Software\Anaconda3\envs\tf_gpu\lib\site-packages\tensorflow\python\client\session.py", line 1429, in _call_tf_sessionrun
    run_metadata)
tensorflow.python.framework.errors_impl.InvalidArgumentError: You must feed a value for placeholder tensor 'Z' with dtype float
        [[{{node Z}}]]
```

图 3-5　计算图中节点的运算结果

了一个计算范式，而只有当运行时根据具体输入值得到具体输出。当尝试仅给定 X 与 Y 运行 result1 和 result2 节点时，程序会报错，因为 result2 需要依赖 Z 值，而此时未给定，自然也无法计算其值。

类似地，tf. InteractiveSession()适用于在交互式命令中使用会话，如 shell 或 IPython 中。与 tf. Session()不同的是，当调用了 tf. InteractiveSession()时，即表示开启了交互式会话，此时不需要显式使用 run 方法得到节点运算结果，取而代之的是使用节点的 eval 方法直接得到值，若待运算的节点依赖注入值，则可直接以字典形式传入 eval 方法。同时，也可以使用和 tf. Session 相同的方式（先得到会话变量 sess，再使用其 run 方法运行节点）使用 tf. InteractiveSession，不过这显然违背了交互式会话的设计初衷。图 3-6 说明了如何在 shell 中使用交互式会话，可以看到直接使用 tf. InteractiveSession()即能开启交互式会话，而无须显式的会话变量，eval 的使用方法也与 Session 的 run 方法类似，具体输出结果类似图 3-5 所示，区别在于交互式会话无法同时对多个操作节点求值，而只能使用单个节点/张量的 eval 方法计算。

```
>>> import tensorflow.compat.v1 as tf
...
... X = tf.placeholder(dtype=tf.float32, name='X')
... Y = tf.placeholder(dtype=tf.float32, name='Y')
... Z = tf.placeholder(dtype=tf.float32, name='Z')
...
... result1 = X + Y
... result2 = result1 * Z

>>> tf.InteractiveSession()

>>> result1.eval(feed_dict={X:1, Y:2})
1.0

>>> result2.eval(feed_dict={X:3, Y:4})
Traceback (most recent call last):
...
tensorflow.python.framework.errors_impl.InvalidArgumentError: 2 root error(s) found.
  (0) Invalid argument: You must feed a value for placeholder tensor 'Z' with dtype float
      [[node Z (defined at <stdin>:6) ]]
      [[mul/_11]]
  (1) Invalid argument: You must feed a value for placeholder tensor 'Z' with dtype float
      [[node Z (defined at <stdin>:6) ]]
0 successful operations.
0 derived errors ignored.
...
>>> result2.eval(feed_dict={X:3, Y:4, Z:5})
3.0
```

图 3-6 交互式会话的使用与输出结果

3.3 TensorFlow 中的张量表示

在 TensorFlow 中，有几种常见的张量表示法，分别使用 tf. constant、tf. Variable、tf. placeholder 和 tf. SparseTensor，在此仅对前 3 种进行介绍。

3.3.1 tf.constant

tf.constant 用于在 TensorFlow 中创建常量张量，创建时需要为函数传入常量的值(value，必须指定)、常量的数据类型(dtype，可选)、张量的形状(shape，可选)、张量的名字(name，可选)，以及是否验证张量的形状(verify_shape，可选)。其中只有常量的值是必须指定的，其他参数皆为可选参数。传入的参数有以下几种特殊情形值得注意：

(1) 当 dtype 未指定时，TensorFlow 会根据传入的常量值自动推断最合适的类型，TensorFlow 中的数据类型在 3.4 节会详细说明。

(2) 当未指定形状 shape 时，TensorFlow 也会根据传入的常量值自动推断形状。

(3) 当指定的形状 shape 与传入的常量形状不一致，并且常量值的个数小于指定的 shape 个数时，TensorFlow 会将常量值的最后一个值填充到缺少的 shape 中得到新的常量，并将新的常量 reshape 为传入的 shape。

(4) 当指定了 verify_shape 时，要么不传入 shape 参数，要么传入的 shape 参数必须与传入的常量值形状一致(这样做其实没有必要，因为此时形状是唯一的)。

tf.constant 使用的代码如下：

```
//ch3/tensor_types.py
import tensorflow.compat.v1 as tf

#建立 4 个含有常量值的节点
#const1 传入整型值
const1 = tf.constant(0)
#const2 传入浮点数
const2 = tf.constant(0.0)
#const3 传入含有整型值的 list,tf 会将其自动转换为 const 张量
const3 = tf.constant([0, 1])
#const4 传入含有整型与浮点数的 list,tf 会将其自动转换为相应数据类型的 const 张量
const4 = tf.constant([0, 1.0])

#初始化会话以运行节点
with tf.Session() as sess:
    #分别运行 4 个常量值节点及直接打印节点
    print(sess.run(const1), const1)
    print(sess.run(const2), const2)
    print(sess.run(const3), const3)
    print(sess.run(const4), const4)
```

运行以上程序，可以得到如图 3-7 所示的结果，从结果可以看出，使用会话运行节点能直接得到节点中的值，如果不通过会话运行而是直接打印节点，则会得到一个张量 Tensor，从中可以看出，这个张量/节点的信息，包括名称、形状、数据类型等。const1 传入的值为整型的 0，其被 TensorFlow 自动转换为 int32 类型(TensorFlow 中对于整型数的默认类型)。

类似地，const2 被转换为默认的 float32 类型。const3 由于传入的 list 都是整型数，因此转换得到张量数据类型为 int32，形状为(2,)。同样地，const4 中由于同时存在整型数与浮点数，此时将数据都转换为浮点数，因此张量的数据类型为 float32。

```
0 Tensor("Const:0", shape=(), dtype=int32)
0.0 Tensor("Const_1:0", shape=(), dtype=float32)
[0 1] Tensor("Const_2:0", shape=(2,), dtype=int32)
[0. 1.] Tensor("Const_3:0", shape=(2,), dtype=float32)
```

图 3-7 在 TensorFlow 中创建常量张量

以上程序说明了第 1 种和第 2 种传参情形，即不指定 dtype 或 shape 参数，让 TensorFlow 自动推断数据类型与数据形状，对第 3 种与第 4 种传参的特殊情形的代码如下：

```
//ch3/tensor_types.py
#第 3 种与第 4 种特殊传参情况
#指定的形状 shape 与传入的常量形状不一致，用常量中最后一个值进行填充
#以 0 填充形状为(2, 3)的数组
const5 = tf.constant(0, shape = [2, 3])

#以 1 填充形状为(2, 3)的数组
const6 = tf.constant([0, 1], shape = [2, 3])

#以 1 填充形状为(2, 3)的数组
const7 = tf.constant([[0], [1]], shape = [2, 3])

#指定的常量形状大于 shape 参数，报错
#const8 = tf.constant([0, 1], shape = [1, 1])

#指定 verify_shape 参数
#指定 verify_shape 为 True 并且常量值形状与给定的 shape 相同
const9 = tf.constant([[0, 1]], shape = [1, 2], verify_shape = True)

#指定 verify_shape 为 True 并且常量值形状与给定的 shape 不同，报错
#const10 = tf.constant([[0, 1]], shape = [2, 1], verify_shape = True)

#指定 verify_shape 为 False 并且常量值形状与给定的 shape 相同
const11 = tf.constant([[0, 1]], shape = [2, 1], verify_shape = False)

with tf.Session() as sess:
    print(sess.run(const5), const5)
    print(sess.run(const6), const6)
    print(sess.run(const7), const7)
    #print(sess.run(const8), const8)
    print(sess.run(const9), const9)
    #print(sess.run(const10), const10)
    print(sess.run(const11), const11)
```

运行以上程序，可以得到如图 3-8 所示的结果，从结果可以看出，只有当传入常量中数字的数量小于 shape 中数字总数时，TensorFlow 才会使用常量中最后一个值进行扩充，否则会报错，其本质相当于先将传入常量 reshape 为(1,)，即一行数字，将这行数中的最后一个数填充满 shape 指定的数字总数，再将这一行数 reshape 为指定的 shape。而 verify_shape 参数为 True 时指定了传入常量值形状必须与传入的 shape 一致。

```
[[0 0 0]
 [0 0 0]] Tensor("Const_4:0", shape=(2, 3), dtype=int32)
[[0 1 1]
 [1 1 1]] Tensor("Const_5:0", shape=(2, 3), dtype=int32)
[[0 1 1]
 [1 1 1]] Tensor("Const_6:0", shape=(2, 3), dtype=int32)
[[0 1]] Tensor("Const_7:0", shape=(1, 2), dtype=int32)
[[0]
 [1]] Tensor("Const_8:0", shape=(2, 1), dtype=int32)
```

图 3-8　tf. constant 的特殊传参情形

3.3.2　tf. Variable

在 TensorFlow 中，使用 tf. Variable 定义变量，变量常用于存储与更新神经网络中的参数。使用 tf. Variable 创建变量时，与 tf. constant 类似，也需要为其传入初始值、形状与数据类型等参数。比较特殊的一点是，tf. Variable 定义的变量还可以传入 trainable 参数，表示定义的该变量是否可训练，其默认值为 True。对于神经网络模型中的参数，其都是可训练的，而对于一些辅助的变量，如迭代计数器等，则是不可训练的。

神经网络模型中的权重通常使用随机数进行初始化，TensorFlow 中也提供了相应的随机数产生函数，如从均匀分布产生随机数使用 tf. random_uniform，使用 tf. random_normal 产生服从正态分布的随机数等。与 tf. constant 不同的是，在用会话运行由 tf. Variable 定义的变量之前，需要先初始化所有定义的变量，使用会话运行 tf. global_variables_initializer()以完成所有变量的初始化之后，才可使用会话运行定义的变量节点，否则会报错“尝试使用未初始化的变量值”。

除了需要先将定义的节点初始化以外，tf. Variable 与 tf. constant 使用方法十分类似，在 TensorFlow 中创建变量的代码如下：

```
//ch3/tensor_types.py
#定义初始化值为 0 的变量,其类型为整型,并定义变量名为 var1
var1 = tf.Variable(initial_value=0, name='var1')

#定义初始化值为[0., 1]的变量,其类型为浮点型,并定义变量名为 var2
var2 = tf.Variable(initial_value=[0., 1], name='var2')

#使用随机正态分布值(均值 1.0,标准差 0.2)初始化变量,形状为(1, 2),并定义变量名为 var3
var3 = tf.Variable(
```

```
        initial_value = tf.random_normal(shape = [1, 2], mean = 1.0, stddev = 0.2),
        name = 'var3'
    )

#使用整型值 10 初始化变量,指定该变量不可训练,并定义变量名为 var4
var4 = tf.Variable(initial_value = 10, trainable = False, name = 'var4')

with tf.Session() as sess:
    #初始化所有定义的变量
    sess.run(tf.global_variables_initializer())
    print(sess.run(var1), var1)
    print(sess.run(var2), var2)
    print(sess.run(var3), var3)
    print(sess.run(var4), var4)

#使用 tf.trainable_variables()打印所有可训练变量
for v in tf.trainable_variables():
    print(v)
```

运行以上程序,可以得到如图 3-9 所示的结果,从结果可以看出,定义的 4 个变量 var1～var4 的值都被正常打印出来了,并且直接打印节点时,可以看到其相应的属性,如名称、形状、数据类型等。读者可以尝试不使用 sess.run(tf.global_variables_initializer())运行变量初始化器,查看直接使用会话运行变量节点时程序的报错情况。

```
0 <tf.Variable 'var1:0' shape=() dtype=int32_ref>
[0. 1.] <tf.Variable 'var2:0' shape=(2,) dtype=float32_ref>
[[1.0481267 0.6772984]] <tf.Variable 'var3:0' shape=(1, 2) dtype=float32_ref>
10 <tf.Variable 'var4:0' shape=() dtype=int32_ref>
<tf.Variable 'var1:0' shape=() dtype=int32_ref>
<tf.Variable 'var2:0' shape=(2,) dtype=float32_ref>
<tf.Variable 'var3:0' shape=(1, 2) dtype=float32_ref>
```

图 3-9 使用 tf.Variable 创建变量并打印结果

以上程序还使用了 tf.trainable_variables 方法得到所有可训练变量并将其一一打印出来,能够发现结果中并没有 var4(其 trainable 参数被指定为 False),说明使用 tf.Variable 定义的变量默认都是可训练的。

3.3.3 tf.placeholder

在 TensorFlow 中,除了使用 tf.constant 创建常量与使用 tf.Variable 创建变量,还可以使用 tf.placeholder 为变量创建占位符。占位符(placeholder),顾名思义,它会为你占据一个位置而不知其具体值。由于计算图仅定义一个计算的范式,而不依赖具体值,所以在搭建计算图的过程中使用占位符作为输入数据的接口是十分常见的做法,当需要运行计算图时再为占位符输入具体值以得到具体的结果。这一点在 3.2 节介绍计算图时也有提及。

与前面几节提到的创建常量与变量类似,使用 tf.placeholder 创建占位符时也有数据类

型(dtype)、形状(shape)与名称(name)等参数可以指定,其中数据类型是必须指定的,因为占位符不像 constant 与 Variable 能自动推断数据类型。不同的是,占位符由于不产生具体值,因此其也没有 value、verify_shape 等参数。值得注意的是,若创建占位符时未指定 shape,则在运行计算图时可以为此占位符传入任意形状的数据。

tf.placeholder 在不同参数下用法的代码如下:

```
//ch3/tensor_types.py
#定义一个数据类型为 float32 的占位符,形状任意
plh1 = tf.placeholder(dtype = tf.float32)
plh2 = tf.placeholder(dtype = tf.float32, name = 'plh2')

#定义一个数据类型为 float32 的占位符,形状为(2, 2)
plh3 = tf.placeholder(dtype = tf.float32, shape = [2, 2], name = 'plh3')

#定义一个数据类型为 float32 的占位符,形状的第一维任意,第二维为 2
plh4 = tf.placeholder(dtype = tf.float32, shape = [None, 2], name = 'plh4')

#定义一个数据类型为 float32 的占位符,形状的第一维任意,第二维也任意
plh5 = tf.placeholder(dtype = tf.float32, shape = [None, None], name = 'plh5')

with tf.Session() as sess:
    print(sess.run(plh1, feed_dict = {plh1: 1}))
    print(sess.run(plh1, feed_dict = {plh1: [1, 2]}))
    print(sess.run(plh1, feed_dict = {plh1: [1, 2, 3]}))
    print(sess.run(plh2, feed_dict = {plh2: 2}))
    print(sess.run(plh3, feed_dict = {plh3: [[1, 2], [3, 4]]}))

    #报错,因为 feed 的数据形状与 placeholder 定义的形状不一致
    #print(sess.run(plh3, feed_dict = {plh3: [[1, 2]]}))

    print(sess.run(plh4, feed_dict = {plh4: [[1, 2]]}))
    print(sess.run(plh4, feed_dict = {plh4: [[1, 2], [3, 4]]}))
    print(sess.run(plh5, feed_dict = {plh5: [[1, 2]]}))
    print(sess.run(plh5, feed_dict = {plh5: [[1, 2], [3, 4]]}))
    print(sess.run(plh5, feed_dict = {plh5: [[1], [2]]}))
```

代码中定义的 plh1 未指定 shape,因此在传入值的时候可以传入任意形状的数据,plh3 指定了数据形状为(2, 2),因此传入的数据形状严格限定为(2, 2),plh4 指定的 shape 为(None, 2),第一维指定为 None 表示该维度任意,只需数据的第二维长度为 2。类似地,可以为 plh5 传入任意形状的二维数据,需要注意的是,plh5 与 plh1 的区别在于,plh1 可以接收任意维度的数据,而 plh5 只能接收维度大小任意的二维数据。

3.4 TensorFlow 中的数据类型

本节对 TensorFlow 中数据类型进行一个简单的介绍。和 Numpy 中类似(参见 2.1.1 节),TensorFlow 中也有自己定义的数据类型,其所有的类型如表 3-1 所示。

表 3-1 TensorFlow 中的数据类型

数据类型	含义	说明
tf.float16	16 位浮点数	
tf.float32	32 位浮点数	
tf.float64	64 位浮点数	
tf.bfloat16	16 位截断浮点数	仅在 TPU 上有原生支持,由 tf.float32 截断前 16 位得到。其表示范围与 tf.float32 相同,但是占用空间仅有其一半。不容易溢出
tf.complex64	64 位复数	
tf.complex128	128 位复数	
tf.int8	8 位有符号整数	
tf.int16	16 位有符号整数	
tf.int32	32 位有符号整数	
tf.int64	64 位有符号整数	
tf.uint8	8 位无符号整数	
tf.uint16	16 位无符号整数	
tf.uint32	32 位无符号整数	
tf.uint64	64 位无符号整数	
tf.bool	布尔型	
tf.string	字符串	
tf.qint8	量化操作的 8 位有符号整数	量化表示将具有连续范围的 float 值以定点近似(如整型)表示。在保证精度近似时压缩模型体积
tf.quint8	量化操作的 8 位无符号整数	
tf.quint16	量化操作的 16 位无符号整数	
tf.qint32	量化操作的 32 位有符号整数	
tf.resource	可变资源值	本书不使用
tf.variant	任意类型值	本书不使用

虽然 TensorFlow 中有这么多不同的数据类型,但是在实际 TensorFlow 编程中,使用 32 位的数据类型居多(浮点数与整数都采用 32 位表示),这也是 TensorFlow 中默认的数字类型格式,除此以外,使用较多的还有布尔型(tf.bool)及字符串(tf.string)。本书不涉及量化操作的数据类型、tf.resource 及 tf.variant 的使用。

在创建张量时,为 dtype 参数传入表 3-1 中的数据类型即可。在实际使用中,计数器性质的变量等使用 tf.int32,而对于输入的图像数据,由于一般将归一化后的数据(像素值大

多落在0～1之间或－1～1之间)作为输入,因此一般使用tf.float32,若直接使用没有归一化的图像,则使用tf.uint8即可。tf.string用于接收字符串类型的变量,一般用于文件或图像路径,得到路径后在计算图中再进行数据读取。

读者可能会有疑问,既然Python及Numpy已经有一套自己的数据类型,为什么TensorFlow还需要定义自己的一套数据类型,这是因为在神经网络训练过程中,不仅仅需要在前向过程中的数值运算,更重要的是反向过程中的模型参数优化过程,这涉及求导等运算过程。因此为了配合TensorFlow中的各种运算操作,定义自己的一套适用于这些操作的数据类型也是必不可少的。同时,由于TensorFlow的大多数据类型与NumPy数据类型直接兼容,因此只需要在为网络输入数据时,直接传入NumPy处理好的数据即可(大多数情况这样做,也可以传入字符串而不直接传入数据),此时模型的输入层会根据数据类型的对应关系直接将NumPy数据转换为兼容的TensorFlow类型的数据,进而完成之后的模型运算。在模型输出时,由于输出结果已不需要进行网络前向与反向计算过程,因此不需要使用TensorFlow中的内置数据类型,返回的结果为NumPy中的数据类型。

与NumPy类似,TensorFlow使用tf.cast方法完成数据类型之间的转换,其具体使用方法为tf.cast([待转换变量], dtype=[目标转换类型]),tf.cast用法的代码如下:

```
//ch3/data_type.py
import tensorflow.compat.v1 as tf

var1 = tf.Variable(1.5, name = 'var1')
#将浮点数转换为int32
re1 = tf.cast(var1, dtype = tf.int32, name = 'var2')

const1 = tf.constant(False, name = 'const1')
#将布尔值转换为float32
re2 = tf.cast(const1, dtype = tf.float32, name = 'const2')

plh1 = tf.placeholder(dtype = tf.string, name = 'plh1')
#将string转换为bool(报错)
re3 = tf.cast(plh1, dtype = tf.bool)

with tf.Session() as sess:
    #初始化所有的变量
    sess.run(tf.global_variables_initializer())
    print(sess.run(var1))
    print(sess.run(re1))

    print(sess.run(const1))
    print(sess.run(re2))

    print(sess.run(plh1, feed_dict = {plh1: 'TensorFlow is awesome'}))

    #报错,不允许从string转换为bool
    #print(sess.run(re3, feed_dict = {plh1: 'TensorFlow is great'}))
```

运行以上程序，可以得到如图 3-10 所示的结果，可以看到原本值为 1.5 的浮点数被转换为整型后值为 1，而布尔值 False 转换为浮点数时变成了 0.0，从此图可以看出，TensorFlow 数据类型之间的转换规则与 Python 和 NumPy 中一致，这也方便了用户的操作。

```
1.5
1
False
0.0
TensorFlow is awesome
```

图 3-10 使用 tf.cast 转换张量的类型

3.5 TensorFlow 中的命名空间

在 TensorFlow 中能十分方便地定义命名空间，使用命名空间有两大好处：一是可以让代码结构更加清晰，使用 TensorBoard 可视化计算图时更加清晰（将在 3.9 节说明 TensorBoard 的用法）。二是通过定义不同的命名空间可以将不同的变量分隔开，这样有利于变量的区分与重用。

在 TensorFlow 中，有两种方式定义命名空间，分别是 tf.name_scope 与 tf.variable_scope，两者在绝大多数情况下是等价的，只有在使用 tf.get_variable 函数时有细微的区别，下面就 tf.get_variable 函数和两种定义命名空间的方式加以说明。

3.5.1 tf.get_variable

如 3.3 节第 2 部分所讲，在 TensorFlow 中可以使用 tf.Variable 定义变量，事实上 tf.get_variable 函数也是定义变量的一种方法。该函数具体参数与 tf.Variable 类似，不过表现形式不同，使用 tf.Variable 创建变量时传入的变量名称 name（可选）是将创建的新变量命名为 name，而使用 tf.get_variable 传入的名称 name 是用来查询是否已经存在名为 name 的变量，如果有则直接返回该变量，否则该函数将创建一个名为 name 的新变量。tf.get_variable 使用方法的代码如下：

```
//ch3/name_scope.py
import tensorflow.compat.v1 as tf

#使用 tf.Variable 创建一个浮点型变量
var1 = tf.Variable(1.2, name = 'var1')

#尝试使用 tf.get_variable 方法获取定义过的变量 var1
var2 = tf.get_variable(name = 'var1', shape = [])

#查看 var2 与 var1 是否指向同一变量
print(var1 == var2)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(var1, sess.run(var1))
    print(var2, sess.run(var2))
```

运行以上程序，读者可以得到 var1==var2 的结果为 False，这说明 var2 与 var1 实际上并不指向同一变量，并且使用会话运行得到的两个变量值也不同，从打印的节点信息可以看出，使用 tf.get_variable 并没有得到已经定义的名为 var1 的变量，而是自动创建了一个新变量，其名为 var1_1。这是因为若想使用 tf.get_variable 重复使用变量或者得到已经创建的变量，其变量也必须是通过 tf.get_variable 创建的，而不可以是通过 tf.Variable 创建的变量，重用变量的写法将在下面讲解命名空间的时候说明。值得注意的是，使用 tf.Variable 总能够创建新的变量，即使传入的 name 是一样的，此时 TensorFlow 会自动解决命名冲突，而使用 tf.get_variable 则严格按照传入的 name 寻找或创建变量。

3.5.2 tf.name_scope

从设计上来讲，tf.name_scope 一般用于操作节点而不用于变量节点，使用 tf.name_scope 定义命名空间时，需要传入名称 name 作为空间名称。当 tf.name_scope 定义的命名空间内存在变量节点时，需要分情况进行说明。当命名空间内有由 tf.Variable 定义的变量时，空间名称会以前缀的形式加在变量名之前，其结构如 scope_name/var_name 所示，命名空间允许嵌套，即结构可以为 scope_name1/scope_name2/…/var_name，这一点与操作节点的表现形式相同，例如空间内的加操作会被命名为类似 scope_name/add 的形式。而使用 tf.get_variable 得到的变量则不受 tf.name_scope 的影响，其参数中指定的 name 即最终得到的变量名称。验证这两者不同之处的代码如下：

```
//ch3/name_scope.py
#定义一个名为 scope1 的命名空间
with tf.name_scope('scope1'):
    #使用 tf.Variable 定义变量 var3
    var3_1 = tf.Variable(2.5, name = 'var3')

    #使用 tf.get_variable 定义变量 var4
    var4_1 = tf.get_variable(name = 'var4', initializer = 0.0)

    #定义加操作
    var5_1 = var3_1 + var4_1

#打印空间内节点信息以查看其名称
print(var3_1)
print(var4_1)
print(var5_1)
```

运行以上程序，可以得到如图 3-11 所示的结果，从结果可以看出，在命名空间中，由 tf. get_variable 定义的变量名不受空间名的影响，而由 tf. Variable 定义的变量名和操作节点都会受其影响。

```
<tf.Variable 'scope1/var3:0' shape=() dtype=float32_ref>
<tf.Variable 'var4:0' shape=() dtype=float32_ref>
Tensor("scope1/add:0", shape=(), dtype=float32)
```

图 3-11 使用 tf. name_scope 创建命名空间

使用 tf. name_scope 并不能完成变量的重用，其作用仅为操作等节点加上空间前缀，使计算图结构更加清晰。在 TensorFlow 中，重用(reuse)属性默认为关闭的，只有通过手动将该属性置为 True 或自动(tf. AUTO_REUSE)才能完成变量的重用，这个属性只存在于 tf. variable_scope 而在 tf. name_scope 不存在。

3.5.3 tf. variable_scope

与 3.5.2 节的 tf. name_scope 类似，tf. variable_scope 也旨在定义计算图的结构，但是区别在于它对于 tf. Variable 和 tf. get_variable 得到的变量都产生作用(为其 name 前加上空间名前缀)，并且其还有 reuse 属性，以完成变量的重用。tf. variable_scope 用法的代码如下：

```
//ch3/name_scope.py
#使用 tf.variable_scope 创建命名空间
with tf.variable_scope('scope1'):
    #使用 tf.Variable 创建名为 var3 的变量
    var3_2 = tf.Variable(3.5, name = 'var3')
    #使用 tf.get_variable 得到名为 var4 的变量
    var4_2 = tf.get_variable(name = 'var4', initializer = 1.0)

#打印命名空间中的变量
print(var3_2)
print(var4_2)
```

运行以上程序，可以得到如图 3-12 所示的结果，从结果可以看出，使用 tf. Variable 和 tf. get_variable 得到的变量名称都受 tf. variable_scope 的影响，需要注意的是 3.5.2 节由 tf. name_scope 定义的 scope1 中已经有名为 var3 的变量，因此 TensorFlow 在此自动为 tf. variable_scope 中的 var3 解决命名冲突，并将其命名为“scope1_1/var3”。

```
<tf.Variable 'scope1_1/var3:0' shape=() dtype=float32_ref>
<tf.Variable 'scope1/var4:0' shape=() dtype=float32_ref>
```

图 3-12 使用 tf. variable_scope 创建命名空间

接下来，我们再使用 tf. variable_scope 定义一个名为 scope1 的命名空间，并尝试在这个命名空间中得到以上代码中定义的 var4_2，这一过程的代码如下：

```
with tf.variable_scope('scope1', reuse = True):
    var4_3 = tf.get_variable(name = 'var4', initializer = 10.0)

print(var4_3 == var4_2)
```

如上述代码所示，新定义的名为 scope1 的命名空间中设置重用(reuse)为 True，使用 tf.get_variable 尝试得到一个名为 var4 的变量。运行以上代码可以得到 var4_3==var4_2 的结果为 True，这说明两者实际上指向的是同一变量，完成对变量 var4_2 的重用。若此时尝试在定义的 scope1 内创建新变量会怎样呢？代码如下：

```
with tf.variable_scope('scope1', reuse = True):
    var4_3 = tf.get_variable(name = 'var4', initializer = 10.0)
    var5_2 = tf.get_variable(name = 'var5', initializer = 100)
```

上述代码尝试在 scope1 内创建新变量 var5_2，var4_3 仍然尝试重用变量 var4_2，此时运行代码会报错，表示其无法找到能够被重用的名为 var5 的变量。因为 scope1 的 reuse 属性指定为 True，这要求命名空间中所有的变量都能够被重用(包括 var5_2，事实上 scope1 中不存在名为 var5 的变量，因此无法重用)。在这种情况下我们希望代码的行为是 var4_3 重用变量 var4_2，而 var5_2 是一个被新建的变量，当在命名空间内变量行为不一致时，我们需要将 reuse 属性置为 tf.AUTO_REUSE，表示是否重用这一行为由 TensorFlow 帮助我们决定，能找到的变量则进行重用，不能找到的就进行新建。正确的写法代码如下：

```
//ch3/name_scope.py
with tf.variable_scope('scope1', reuse = tf.AUTO_REUSE):
    var4_3 = tf.get_variable(name = 'var4', initializer = 10.0)
    var5_2 = tf.get_variable(name = 'var5', initializer = 100)

print(var4_3 == var4_2)
print(var5_2)
```

运行以上代码，可以得到如图 3-13 所示的结果，可以发现此时 var4_3 完成了对变量 var4_2 的重用，同时 var5 也被创建成值为 100(从 dtype 为 int32 可以得知)的变量。笔者建议使用 tf.variable 时尽量将 reuse 设置为 tf.AUTO_REUSE 以防止空间内变量行为不一致的情况。

```
True
<tf.Variable 'scope1/var5:0' shape=() dtype=int32_ref>
```

图 3-13　使用 tf.variable_scope 重用变量

当有嵌套的命名空间时，还可以通过手动指定带有命名空间的变量名完成重用，下面的程序首先嵌套定义命名空间 scope_x 与 scope_y，并在 scope_y 中定义了一个变量 reuse_var，该变量在命名空间的作用下全名变为 scope_x/scope_y/reuse_var。接着定义一个名为

scope_x 的命名空间并指定其中的变量自动进行重用(reuse=tf. AUTO_REUSE),现尝试重用名为 scope_y/reuse_var 的变量。运行程序后,发现 reuse_var2 能够成功重用变量 reuse_var,这说明重用变量时可以手动指定需要重用的命名空间前缀,代码如下:

```
//ch3/name_scope.py
with tf.variable_scope('scope_x'):
    with tf.variable_scope('scope_y'):
        reuse_var = tf.get_variable('reuse_var', initializer=1000.0)

with tf.variable_scope('scope_x', reuse=tf.AUTO_REUSE):
    reuse_var2 = tf.get_variable('scope_y/reuse_var', initializer=0.001)
```

需要注意的是,tf. variable_scope 的 reuse 属性仅对作用域中与 tf. get_variable 相关的变量起作用,这是因为 tf. Variable 仅起到新建变量的作用,而没有获取已有变量的作用,读者可以将上面代码中的 tf. get_variable 改成 tf. Variable 进行尝试。

3.6 TensorFlow 中的控制流

程序中除了最常见的顺序结构,还有分支结构(if、switch 等)与循环结构(for、while 等)等。由于计算图是固定的、静态的,而有时候神经网络模型需要根据模型的不同状态使用动态的分支或者循环结构,因此 TensorFlow 提供了一套专门用于计算图中的控制流操作,用于计算图中的动态结构。

除了结构以外,TensorFlow 还能指定计算图中节点的执行顺序。下面就分别对 TensorFlow 中的分支结构与循环结构及如何指定节点执行顺序进行讲解。

3.6.1 TensorFlow 中的分支结构

在程序设计中,常常使用 if/else 与 switch/case 语句完成分支结构(虽然 Python 中没有 switch/case 语句)。相应地,TensorFlow 也提供了两种语句在计算图中的实现形式,与 if/else 对应的函数为 tf. cond,而与 switch/case 语句对应的函数为 tf. case,下面就分别对这两个函数进行介绍。

与 if 的控制流一样,tf. cond 函数一次只对单个分支条件值进行判断,若该值为 True,则执行某些函数(true_fn),否则执行另一些函数(false_fn)。tf. cond 函数有几个重要的参数,分别为分支条件 pred(True 张量或者 False 张量),条件为 True 时执行的函数 true_fn,条件为 False 时执行的函数 false_fn。需要注意的是,TensorFlow 要求 true_fn 与 false_fn 不可有参数且必须有返回值,并且这两个函数的返回值的数据类型与返回参数个数与结构需要相同,简单来说,若 true_fn 返回两种类型为 float32 的张量,则 false_fn 也必须返回两种类型为 float32 的张量。tf. cond 的具体使用方法的代码如下:

```
//ch3/control_flow.py
import tensorflow.compat.v1 as tf

#定义常量 a 与 b,其值分别为 1.0 与 2.0
a = tf.constant(1.0, name = 'a')
b = tf.constant(2.0, name = 'b')

#定义一个判断条件的占位符,其类型为 tf.bool
condition = tf.placeholder(dtype = tf.bool, name = 'condition')

#当 condition 为 True 时,返回 c = a + b,否则返回 c = a - b,此处使用匿名函数实现
c = tf.cond(condition, lambda: a + b, lambda: a - b)

#当 a < b 时,d = a * b,否则 d = a/b
d = tf.cond(a < b, lambda: a * b, lambda: a / b)

#由于计算图中不存在变量,因此不需要使用 variable_initializer
with tf.Session() as sess:
    #根据传入的不同 bool 值得到不同的结果
    print(sess.run(c, feed_dict = {condition: True}))
    print(sess.run(c, feed_dict = {condition: False}))
    print(sess.run(d))
```

程序中使用 TensorFlow 中的条件语句定义了 c 与 d 的值,运行程序能够发现当传入的 condition 不同时,得到的 c 值也不同。

类似地,tf.case 与 switch/case 的控制流一样(尽管 Python 不提供 switch/case 语句)。tf.cond 函数以键值对的形式传入每个 case 条件,其形式为{case1: func1, case2: func2,…},并且其还有一个 default 参数,表示当没有 case 匹配时需要执行的函数,同时 tf.case 还有一个 exclusive 参数,当 exclusive 为 True 时,表示传入的控制流中至多只能有一个分支为 True,若多于一个条件为 True,则报错。tf.case 的具体使用方法的代码如下:

```
//ch3/control_flow.py
#定义常量 a 与 b,其值分别为 1.0 与 2.0
a = tf.constant(1.0, name = 'a')
b = tf.constant(2.0, name = 'b')

#定义一个判断条件的占位符,其类型为 tf.int32
condition = tf.placeholder(dtype = tf.int32, name = 'condition')

#使用键值对定义 case,并指定 exclusive 为 False
c = tf.case(
    {condition > 1: lambda: a + b, condition > 2: lambda: a + 2 * b},
    default = lambda: a - b, exclusive = False
)
```

```
#使用键值对定义 case,并指定 exclusive 为 True,此时会报错,因为两个条件在 condition > 2 时都
#为 True
d = tf.case(
    {condition > 1: lambda: a + b, condition > 2: lambda: a + 2 * b},
    default = lambda: a - b, exclusive = True
)

#由于计算图中不存在变量,因此不需要使用 variable_initializer
with tf.Session() as sess:
    #根据传入不同的 condition 值得到不同的结果
    print(sess.run(c, feed_dict = {condition: 1}))
    print(sess.run(c, feed_dict = {condition: 2}))
    print(sess.run(c, feed_dict = {condition: 3}))

    print(sess.run(d, feed_dict = {condition: 1}))
    print(sess.run(d, feed_dict = {condition: 2}))
    #报错
    print(sess.run(d, feed_dict = {condition: 3}))
```

运行以上程序,会发现当 exclusive 为 True 并且传入的 condition 为 3 时程序会报错并退出,这是因为当 condition 为 3 时,此时传入的 case 多于一个为 True。细心的读者再运行以上程序会发现程序会抛出一个 WARNING:TensorFlow:case: An unordered dictionary of predicate/fn pairs was provided, but exclusive=False. The order of conditional tests is deterministic but not guaranteed.,这是因为传入的 case 为字典类型,其本身不保证顺序性,若要消除这个 WARNING,只需将传入的 case 改为有确定性顺序的列表,将传入的 case 改成如下"列表+元组"的形式即可,代码如下:

```
c = tf.case(
    [(condition > 1, lambda: a + b), (condition > 2, lambda: a + 2 * b)],
    default = lambda: a - b, exclusive = False
)
```

3.6.2 TensorFlow 中的循环结构

在 TensorFlow 中,使用 tf.while_loop 完成循环,其参数需要传入循环条件函数、循环体函数及这两个函数需要传入的参数,tf.while_loop 的返回值与循环体函数的返回值形式保持一致,并且条件判断函数与循环体函数的参数列表需要保持一致,因为在循环体函数中有可能对判断条件的参数进行了更改,因此需要循环体函数接收条件判断函数所有的参数,同时要求循环体参数将传入的所有参数进行返回。下面的程序说明了 tf.while_loop 的用法,实现了一个变量加 1 的操作,代码如下:

```
//ch3/control_flow.py
#定义循环中需要使用的变量 i 和 n
i = 0
n = 10

#循环条件函数
def judge(i, n):
    #当 i < sqrt(n)时才执行循环
    return i * i < n

#循环体函数
def body(i, n):
    #循环中使 i 增 1
    i = i + 1

    #返回的参数与输入的参数保持一致
    return i, n

#为 tf.while_loop 传入条件函数、循环体函数及参数
new_i, new_n = tf.while_loop(judge, body, [i, n])

with tf.Session() as sess:
    print(sess.run([new_i, new_n]))
```

运行以上程序，可以得到最终由循环得到的 new_i 与 new_n 分别为 4 和 10，说明当 i 由 0 增加到 4 时，由于 $4\times4>10$ 而跳出循环得到最终结果。

3.6.3 TensorFlow 中指定节点执行顺序

TensorFlow 中可以对计算图中的节点指定执行顺序，这可以使用 tf.control_dependencies 进行实现，首先需要明确 tf.control_dependencies 会创建一个作用域，创建该作用域时需要为 tf.control_dependencies 传入一个操作或者计算图中节点的列表，表示这个列表中的操作需要先于作用域中的操作执行。tf.control_dependencies 用法的代码如下：

```
//ch3/control_flow.py
#定义一个变量 x,其初始值为 2
x = tf.Variable(2)
#为 x 定义一个加 1 操作
x_assign = tf.assign(x, x + 1)
#y1 的值为 x^2
y1 = x ** 2

with tf.control_dependencies([x_assign]):
    #y2 的值为 x^2,其需要在执行 x_assign 之后执行
```

```
    y2 = x ** 2

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    print(sess.run([y1, y2]))
```

运行程序后，可以得到 y_1 的值为 4，而 y_2 的值为 9，这说明 y_1 在 x 未提供加 1 时即得到了计算，而 y_2 由于有控制依赖，它的计算在 x_assign(即 x 已经加 1)后才被执行。如上代码的计算图可以用图 3-14 进行表示。

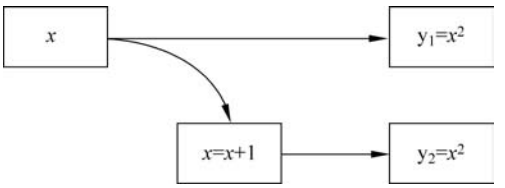

图 3-14 含有 tf.control_dependencies 的计算图

3.7 TensorFlow 模型的输入与输出

明确了 TensorFlow 中的基本概念(如张量、计算图等)，想要理解 TensorFlow 中对模型的输入与输出便也不难了。

在 TensorFlow 中，最为常见的模型输入便是使用 tf.placeholder 进行实现，在运行计算图时，再为该占位符输入具体的训练数据。由于在训练网络时，常常因为训练数据过多过大而无法一次性全部放入内存，所以在训练模型时可以将数据分为一个个小的 batch 放入网络进行训练，这也恰好满足了 tf.placeholder 的特性，在每一次训练迭代时将不同的 batch 数据放入占位符以完成训练。

模型的输出通常为 NumPy 类型，在使用 tf.Session 得到模型输出的具体结果后可以通过处理 NumPy 数据的一切方法进行后期处理。

3.8 TensorFlow 的模型持久化

训练神经网络模型常常是一件十分耗时的事情，因此如果能将训练好的网络权值保存到磁盘，当需要使用该权值时再从磁盘中进行恢复以继续训练或者仅仅只进行前向推理，将省下每次需要重复训练的大量时间。

在 TensorFlow 中，使用 saver 对模型进行持久化，由 tf.train.Saver 创建。使用 saver 既可以完成对模型的保存，也能完成对磁盘上已保存的权值的读取，分别使用其 save 和 restore 方法。下面就分别介绍如何使用 tf.train.Saver 完成权值的保存与读取。

3.8.1 模型的保存

当定义好计算图时，可以使用 tf. train. Saver 的 save 方法保存计算图中的变量，当使用 tf. train. Saver 创建对象时，可以为其构造函数传入一个 var_list，表示仅保存 var_list 中指定的变量，否则默认保存计算图中所有的变量。在使用 save 方法时，需要传入当前所在的 Session（因为在不同的 Session 下，同一模型的参数值有可能不相同）与需要保存的文件名，下面的程序说明了如何使用 saver 保存计算图中的变量，为简便起见，程序中定义的计算图沿用 3.2 节第 1 部分中定义的 $(X+Y)\times Z$ 的计算图，不同的是将 X 与 Y 改为 tf. Variable 而非 placeholder（因为 saver 仅可保存计算图中的变量，而原计算图中全都为 placeholder，因此无法保存），代码如下：

```
//ch3/handle_ckpt.py
import tensorflow.compat.v1 as tf

X = tf.Variable(56.78, dtype = tf.float32, name = 'X')
Y = tf.Variable(12.34, dtype = tf.float32, name = 'Y')
Z = tf.placeholder(dtype = tf.float32, name = 'Z')

#结果 1 为 X 与 Y 相加
result1 = X + Y
#结果 2 为(X + Y) * Z
result2 = result1 * Z

saver = tf.train.Saver()

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    saver.save(sess, 'graph.ckpt')
```

代码中在 Session 外初始化了一个 saver，并不指定 var_list，表明需要保存计算图中的所有变量，在会话内使用 save 方法将计算图中的变量保存成名为 graph. ckpt 的权重文件。运行以上程序可以发现根目录下多了 4 个文件，其中 3 个分别以 data、index 和 meta 结尾，还有一个为 checkpoint 文件，其中以 data 结尾的文件存储了计算图中变量的具体值，而以 index 结尾的文件提供了文件索引，以 meta 结尾的文件存储了计算图结构而没有任何具体值，checkpoint 以文本形式存储了最新的一个权值名称，在此不涉及这几个文件内部具体的存储方式，下面通过代码说明如何查看这些权值文件中究竟保存了哪些变量及对应的值。

在 TensorFlow 中，可以使用 tf. train. NewCheckpointReader 来查看权值文件中具体存储了哪些变量及其对应的值。使用 debug_string 方法查看权值文件中具体存在的变量名，使用 get_tensor 并传入相应的变量名查看其对应的值。tf. train. NewCheckpointReader 使用方法的代码如下：

```
new_ckpt = tf.train.NewCheckpointReader('graph.ckpt')
print(new_ckpt.debug_string().decode('utf8'))
print(new_ckpt.get_tensor('X'), new_ckpt.get_tensor('Y'))
```

运行以上程序,可以得到如图 3-15 所示的结果。从结果可以看出,权值文件中保存了两个变量,其名称分别为 X 和 Y,类型为 float,并且形状为空(表明是标量),X 和 Y 的值分别为 56.78 和 12.34,这和前面定义计算图过程中的变量值完全一致。

```
X (DT_FLOAT) []
Y (DT_FLOAT) []

56.78 12.34
```

图 3-15 查看权值文件中的变量及其值

3.8.2 模型的读取

通过 3.8.1 节的学习,我们已经成功保存了网络中的变量,并通过 NewCheckpointReader 查看了权值文件中的变量名与值。本节对权值文件进行读取,将值恢复到模型中,并进一步进行操作。

在 TensorFlow 中,恢复模型权值使用 saver 的 restore 方法。同样,在创建 saver 时可以传入一个 var_list 以指定需要被恢复的变量,restore 方法需要传入目标 Session 和磁盘上权值文件的路径。在进行恢复时,我们常常会读取一个路径下最新保存的权值进行恢复(默认认为最终的权值文件是最好的),这一过程常常使用 tf.train.latest_checkpoint 来完成,为该函数传入权值文件所在的文件夹,它就会自动得到最新的权值文件路径。读取权值文件中的权重并将其恢复的代码如下:

```
//ch3/handle_ckpt.py
#重新定义计算图,并改变变量的初始值
X = tf.Variable(11.1111, dtype=tf.float32, name='X')
Y = tf.Variable(22.2222, dtype=tf.float32, name='Y')
Z = tf.placeholder(dtype=tf.float32, name='Z')

#结果 1 为 X 与 Y 相加
result1 = X + Y
#结果 2 为(X + Y) * Z
result2 = result1 * Z

var_list = [X]
saver = tf.train.Saver(var_list=var_list)

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    last_ckpt = tf.train.latest_checkpoint('.')
    print(last_ckpt)
    saver.restore(sess, last_ckpt)
```

```
    print(sess.run(X))
    print(sess.run(Y))
```

运行以上程序，可以得到 sess.run(*X*)结果为 56.78 而非定义的 11.1111，这是因为在调用 tf.global_variables_initializer 对 *X* 进行初始化后，又使用 saver 将权值文件中保存的 *X* 值(56.78)恢复到变量 *X* 中了，而 sess.run(*Y*)的值则是 22.2222，与代码中定义的 *Y* 值相同，说明 *Y* 值没有被权值文件中恢复的值覆盖，因为代码中指定了 var_list 中仅含有 *X* 变量，所以变量 *Y* 的值仍是初始化的值 22.2222。

3.9 使用 TensorBoard 进行结果可视化

TensorBoard 是 TensorFlow 的可视化工具包，使用 TensorBoard 能够可视化实时跟踪模型训练过程中的数据量变化情况，如训练损失等。它还能对图像及计算图等进行可视化。TensorBoard 是 TensorFlow 框架的一大优势，所以本节介绍 TensorBoard 可视化的用法。

为了保存计算图中的各种信息，首先需要使用 tf.summary.FileWriter 初始化一个 writer 对象，表示使用其写入网络中的各种信息。

3.9.1 计算图的可视化

为了简便起见，本节使用的是 3.8 节第 1 部分所定义的计算图，为了保存代码中定义的计算图，只需要在定义 writer 时为 graph 参数传入当前的默认计算图(tf.get_default_graph()得到当前默认的计算图)，在程序结束前使用 writer.close 关闭这个 IO 对象即可(纯粹是一个好习惯)。下面的程序说明了如何将计算图添加到 TensorBoard 文件中，其中加粗的代码与 TensorBoard 操作直接相关，代码如下：

```
//ch3/use_tb.py
import tensorflow.compat.v1 as tf

X = tf.Variable(56.78, dtype = tf.float32, name = 'X')
Y = tf.Variable(12.34, dtype = tf.float32, name = 'Y')
Z = tf.placeholder(dtype = tf.float32, name = 'Z')

#结果 1 为 X 与 Y 相加
result1 = X + Y
#结果 2 为(X + Y) * Z
result2 = result1 * Z

#创建一个 summary 的 IO 对象，并将计算图添加到 summary 中
writer = tf.summary.FileWriter('summary', graph = tf.get_default_graph())

with tf.Session() as sess:
```

```
    tf.global_variables_initializer().run()

#关闭 IO 对象
writer.close()
```

运行以上程序，会发现根目录下多了一个 summary 文件夹，其中有一个名为 events.out.tfevents…的文件。此时在 summary 文件夹下打开命令行，使用 TensorBoard -logdir . 指令，并在浏览器中输入 localhost:6006（6006 是 TensorBoard 默认使用的端口，当然可以更改），此时可以看到如图 3-16 所示的结果。

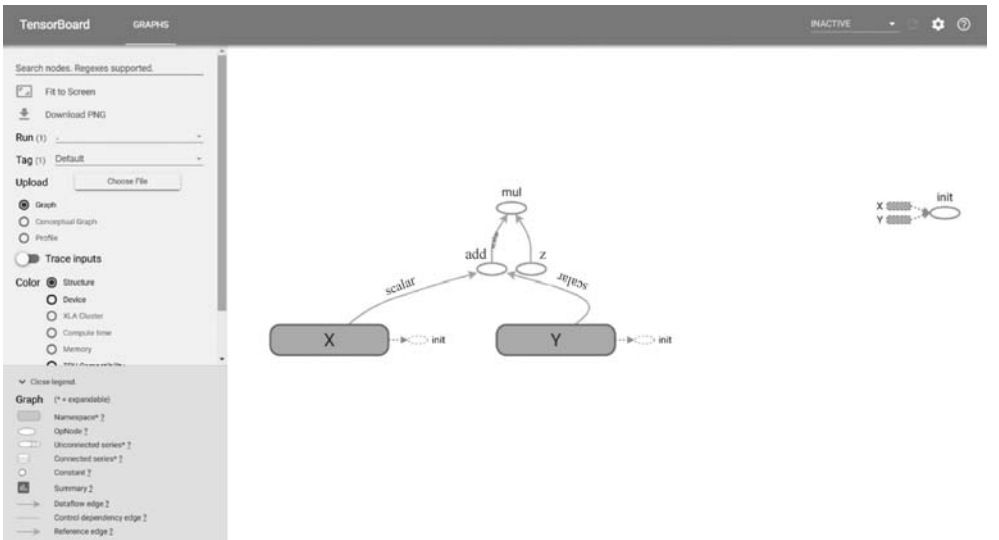

图 3-16 使用 TensorBoard 查看计算图的结构

从结果可以看出，使用 TensorBoard 画出来的计算图与我们在 3.2 节第 1 部分人工绘制的计算图结构一致。不同的是，TensorBoard 绘制的计算图更加细致，如果读者尝试双击 X 或 Y 节点，则可以看到节点内部更加细化的操作（如初始化操作等）。

3.9.2 矢量变化的可视化

在训练模型时，常常需要保存与查看损失函数的变化情况。此时可以使用 TensorBoard 保存损失以查看损失随着迭代次数的变化情况。在 TensorFlow 中，使用 tf.summary.scalar 保存需要记录的标量，为该函数传入名称与对应的张量即可。下面的程序使用了循环完成变量的加 1 操作，并使用 TensorBoard 记录了变量的变化过程，代码如下：

```
//ch3/use_tb.py
i = tf.Variable(1)
```

```
writer = tf.summary.FileWriter('summary_1', graph=tf.get_default_graph())

assign_op = tf.assign(i, i + 1)

tf.summary.scalar('i', i)
merged_op = tf.summary.merge_all()

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for e in range(100):
        sess.run(assign_op)
        summ = sess.run(merged_op)
        writer.add_summary(summ, e)
writer.close()
```

在以上代码中，除了使用了 tf.summary.scalar 来记录张量 i 以外，还使用了 tf.summary.merge_all 函数，由于定义的模型可能十分庞大(当然示例代码的计算图很小)，模型的各部分都在不同的模块中，此时对于模型的描述(summary)可能分布在各个文件，此时就需要使用 tf.summary.merge_all 将“散落”在各部分的 summary 收集起来使其变成一个操作，也就是代码中的 merge_op。得到总的 merge_op 后再使用 Session 得到描述的具体值，再将该具体值放入 IO 对象 writer(writer.add_summary(sum, e))。使用 writer.add_summary 时需要注意，除了需要给其传入具体的 summary 以外，还需要给当前这个 summary 绑定一个周期 e，表示这个 summary 中的值对应于第 e 个周期的数据。如以上代码，整体使用一个 for 循环了 100 个周期，在每个周期内，先用 Session 运行加 1 操作，再得到加 1 操作后计算图中的 i 变量值，最后将得到的 i 值与当前周期 e 绑定放入 writer。

运行以上程序后，并在根目录的 summary_1 文件夹下使用 TensorBoard -logdir . 命令，可以得到如图 3-17 所示的结果，可以看到此时有两个选项 SCALARS 和 GRAPH，在 GRAPH 选项卡中可以看到我们定义的计算图，在 SCALARS 里可以看到我们记录的变量 i，并且可以发现其值从 2 一直增加到 101(读者可以思考为什么不是从 1 开始增加到 100，以及如何才能使 i 值从 1 增加到 100)。

3.9.3 图像的可视化

在某些应用场景下，我们需要以图像的形式可视化模型中间层的表示或者对于生成式模型而言，我们需要可视化模型最终生成的结果，此时可以使用 TensorBoard 对图像进行可视化，与可视化标量类似，使用 tf.summary.image 函数可以对图像进行可视化，下面的程序说明了使用 TensorBoard 可视化图像的过程，运行程序前需要确保根目录下有一张名为 1.jpg 的图像，代码如下：

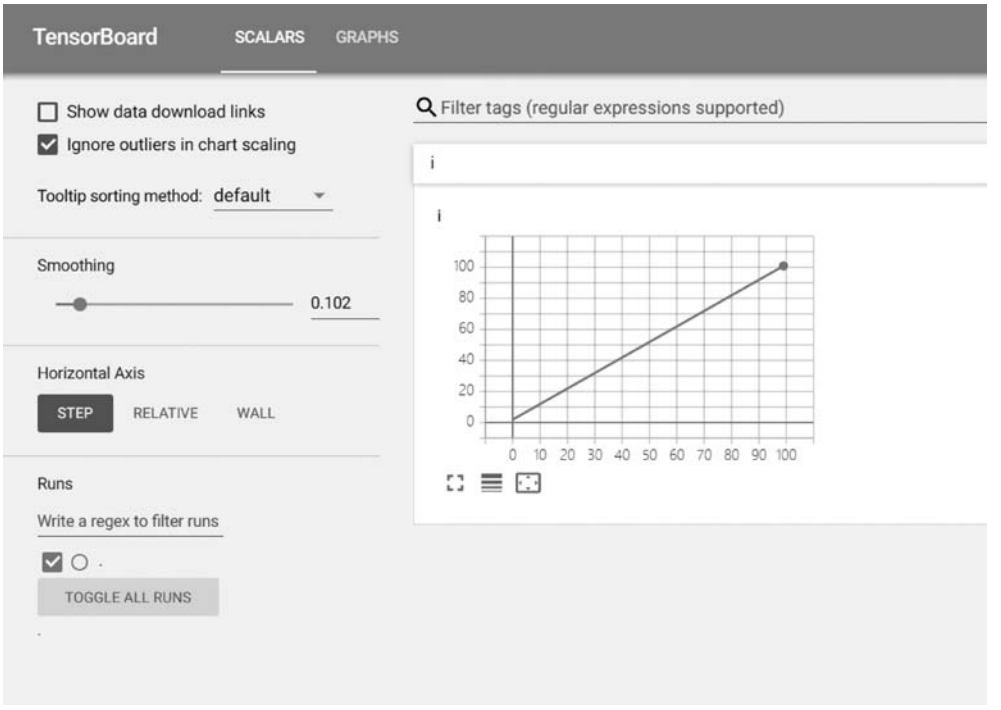

图 3-17 使用 TensorBoard 记录标量变化情况

```
//ch3/use_tb.py
with open('1.jpg', 'rb') as f:
    data = f.read()

#图像解码节点,3 通道 jpg 图像
image = tf.image.decode_jpeg(data, channels = 3)

#确保图像以 4 维张量的形式表示
image = tf.stack([image] * 3)

#添加到日志中
tf.summary.image("image1", image)
merged_op = tf.summary.merge_all()

writer = tf.summary.FileWriter('summary_2', graph = tf.get_default_graph())

with tf.Session() as sess:
    #运行并写入日志
    summ = sess.run(merged_op)
    writer.add_summary(summ)

writer.close()
```

需要注意的是，由于在训练模型时，图像数据常常以一个 batch 的形式放入模型进行训练，一个 batch 数据为一个 4 维张量，其形状为(batch_size，H，W，C)，batch_size 表示 batch 中的图像数量，H、W 和 C 分别表示图像的高度、宽度与通道数。在使用 TensorBoard 可视化图像时，其也要求被记录的图像是一个 4 维张量，TensorBoard 会将张量中的每张图像依次显示出来，一共显示 batch_size 张图像。因此如上代码中，使用 tf.image.decode_jpeg 对单张图像解码后，其形状为(H，W，C)，不符合 TensorBoard 的要求，因此在之后又使用了 tf.stack([image] * 3)将读取的图像堆叠了 3 次，得到的张量形状为(3，H，W，C)，在最后使用 TensorBoard 可视化图像时会得到 3 张相同的图像。

运行以上程序，能够得到如图 3-18 所示的结果，从结果可以看出，3 张相同的图像被依次显示在 image1 的选项卡下。当然，writer.add_summary 此时依旧接收 step 参数，读者可以传入周期数或者迭代数，以方便查看图像随着周期的变化情况。

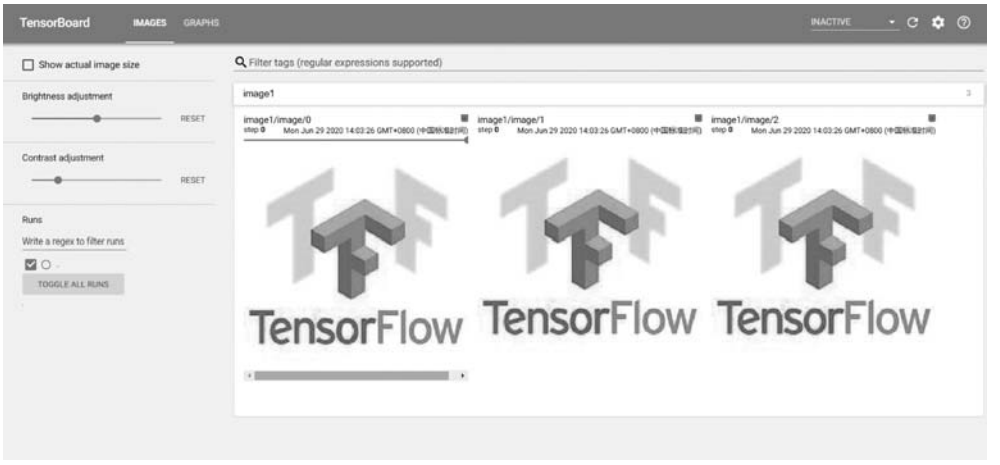

图 3-18　使用 TensorBoard 可视化图像

本节简要说明了 TensorBoard 的用法，分别介绍了使用 TensorBoard 对于计算图、标量及图像的可视化方法，这三者也是使用 TensorBoard 最常用的可视化对象。当然，TensorBoard 实际上还能完成更多复杂数据的可视化，如使用直方图等可视化张量的统计信息，其使用方法与标量或图像的可视化类似，对于复杂数据的可视化本书在此并不涉及，有兴趣的读者可以自行查阅资料进行学习。

3.10　小结

本章就 TensorFlow 的基础知识进行了讲解，从 TensorFlow 最基本的计算图与张量等概念讲到 TensorBoard 在可视化方面的使用。关于 TensorFlow 还有许多高级的函数与用法，读者可以自行到 TensorFlow 官网(https://www.tensorflow.org/)进行学习。