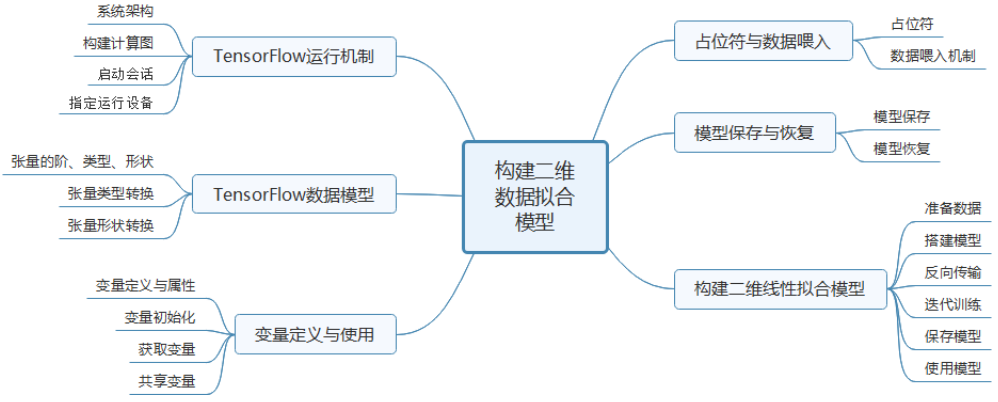
任务 2　构建二维数据拟合模型

本章内容

本章将着重阐述 TensorFlow 的运行机制及语法基础。首先介绍 TensorFlow 中计算图与会话的概念，接着阐述其数据模型，并以案例形式展示 TensorFlow 中的常量、变量、占位符、模型保存、恢复的使用方法，最后以二维数据拟合模型为例，讲解深度学习模型的搭建过程。

知识图谱



重点难点

重点：掌握 TensorFlow 计算图与会话的运行机制，熟练使用张量，并能对张量进行各种运算与形状转换。

难点：数据的喂入机制与张量形状的变换。

2.1　TensorFlow 运行机制

TensorFlow 是一个基于计算图的深度学习编程模型。其名称表示了它的运行原理，Tensor 表示张量，其实质上是某种类型的多维数组；Flow 表示基于数据流图的计算，实质上是张量在不同节点间的转化过程。

在 TensorFlow 中，计算图中的节点称为 OP（即 operation 的缩写），节点之间的边描

述了计算之间的依赖关系。计算过程中，一个节点可获得 0 或多个张量，产生 0 或多个张量。

TensorFlow 程序通常被组织成图的构建阶段和执行阶段。在构建阶段，节点的执行步骤被描述成一个图；在执行阶段，使用会话执行图中的 OP。

2.1.1　TensorFlow 系统架构

TensorFlow 支持各种异构平台，支持多 CPU/GPU、移动设备，具有良好的跨平台的特性；且架构灵活，能支持各种网络模型，具有良好的通用性。系统结构以 C API 为界限，将整个系统分为前端和后端两个子系统（见图 2-1）。前端系统提供编程模型，负责构建计算图，后端系统提供运行时环境，负责在会话中执行计算图。

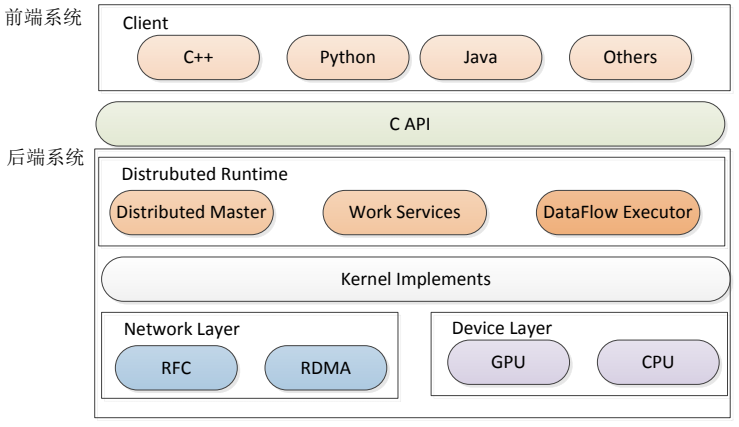


图 2-1　TensorFlow 系统架构

图 2-1 所示组件构成 TensorFlow 系统分布式运行机制的核心，其中每个组件的作用如下。

- Client：是前端系统的主要组成部分。它是一个支持多语言的编程环境，提供了基于计算图的编程模型，方便用户构造各种复杂的计算图，实现各种形式的模型设计。
- Distrubuted Runtime：在分布式的运行时环境中，Distributed Master 根据 Session.run 的 Fetching 参数，从计算图中反向遍历，找到所依赖的最小子图。对于每个任务，TensorFlow 都将启动一个 Worker Service，按照计算图中节点之间的依赖关系，根据当前的可用的硬件环境（CPU / GPU），调用节点的 Kernel 实现完成节点的运算。
- Kernel Implements：大多数 Kernel 基于 Eigen::Tensor 实现。Eigen::Tensor 是一个使用 C++模板技术，为多核 CPU/GPU 生成高效的并发代码，包含 200 多个标准的张量，包括数值计算、多维数组操作、控制流、状态管理等。每一个节点根据

设备类型都会存在一个优化了的 Kernel 实现，在运行时，运行时根据本地设备的类型，为节点选择特定的 Kernel 实现，完成该节点的计算。

2.1.2　构建计算图

计算图描述了一组需要依次序完成的计算单元以及这些计算单元之间相互依赖的关系。图中的节点表示某一具体的计算单元，如张量以及张量之间的乘积、点积或卷积计算等。

计算图的构建阶段也称为计算图的定义阶段，该过程会在图模型中定义所需的运算，每次运算的结果以及原始的输入数据都可称为一个节点。

下面的代码给出了计算图的定义过程（代码位置：chapter02/define_graph.py）。

```
1  import tensorflow as tf
2  a = tf.constant([3,5],dtype=tf.int32)
3  b = tf.constant([2,4],dtype=tf.int32)
4  result = tf.add(a,b)
5  print(result)
```

上述代码定义了两个张量相加的计算图（见图 2-2）。

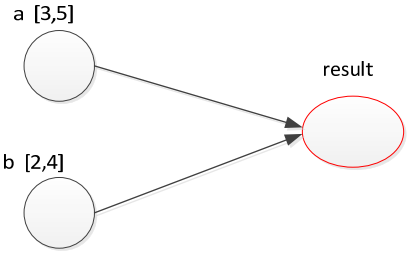


图 2-2　张量相加计算图模型

图 2-2 中，每个节点表示一个运算，每条边代表计算之间的依赖关系。a、b 节点表示两个 1 行 2 列的矩阵，result 节点则依赖于读取 a 和 b 的值，从 a、b 有两条到 result 的边，没有任何计算依赖 result 的结果，result 节点没有指向其他节点的边。

第 1 行代码：导入 tensorflow 类库。后续可使用 tf 代替 tensorflow 作为模块名称，整个程序更加简洁。

第 2 行代码：定义一个整型张量 a，表示 1 行 2 列的矩阵，其值为[3,5]。

第 3 行代码：定义一个整型张量 b，表示 1 行 2 列的矩阵，其值为[2,4]。

第 4 行代码：tf.add() 函数用于对张量 a、b 相加，并将计算结果返回给 result 节点。

第 5 行代码：print(result)函数打印 result 节点的信息。

运行程序，输出结果如下。

```
Tensor("Add:0", shape=(2,), dtype=int32)
```

从运算结果可以看出，上述程序并没有输出矩阵相加的结果，而是输出了一个包含 3 个属性的张量，第 1 个属性表示加法运算，第 2 个属性表示包含 2 个元素的一维数组，第 3 个属性表示数据类型为整型。

需要注意的是，TensorFlow 计算图一般包含两个特殊节点，分别为 Source 节点（也称为 Start 节点）与 Sink 节点（也称为 Finish 节点）。其中，Source 节点表示此节点不依赖于任何其他节点作为输入，Sink 节点表示该节点无任何输出作为其他节点的输入。

2.1.3　在会话中运行计算图

在模型运行环节中，计算图只能在会话提供的上下文环境中启动，并执行相关的运算。会话提供了一定的资源和环境，可以将计算图的节点分发到 CPU 或 GPU 设备上，同时提供执行节点的方法。方法执行后，将产生的张量返回。

在 TensorFlow 中启动会话有两种方法。

（1）明确调用创建会话函数和关闭会话函数，代码格式如下：

```
#创建一个会话
sess =tf.Session()
#使用该会话来运行相关节点，得到运算结果
sess.run(...)
#关闭会话
sess.close()
```

上述代码中，所有计算完成后，需要明确调用 close()函数来关闭会话并释放资源。若程序因为异常退出，就可能无法执行 close()函数，进而导致资源泄露。为了解决异常退出时的资源泄露问题，TensorFlow 中还可以通过上下文管理器来管理会话。

（2）通过上下文管理器来管理会话，代码格式如下：

```
#创建一个会话，并使用上下文管理器管理会话
with tf.Session() as sess:
#使用该会话来运行相关节点，得到运算结果
    sess.run(...)
#不需要调用 close()函数，当上下文退出时会话关闭，资源自动释放
```

下面通过上下文管理器来实现两个张量的相加，代码如下（代码位置：chapter02/perform_graph.py）。

```
1  import tensorflow as tf
2  a = tf.constant([3,5],dtype=tf.int32)
3  b = tf.constant([2,4],dtype=tf.int32)
4  result = tf.add(a,b)
5  print(result)
6  with tf.Session() as sess:
7       print(sess.run(result))
```

第 6 行代码：创建一个会话，并使用上下文管理器管理会话。

第 7 行代码：在会话中调用 session.run(result)，执行计算图的 result 节点，即执行加法运算，并获得输出结果[5 9]。

在 TensorFlow 中，系统会自动维护默认的计算图。通过 tf.get_default_graph() 函数可以获得当前默认的计算图，代码如下（代码位置：chapter02/default_graph.py）。

```
1  import tensorflow as tf
2  a=tf.constant([1.0,2.0], name='a')
3  b=tf.constant([1.0,2.0], name='b')
4  result = a+b
5  print(a.graph is tf.get_default_graph())
```

第 5 行代码：通过 a.graph 查看张量所属的计算图。如果没有特别指定，则查看当前默认的计算图。

除了默认计算图，TensorFlow 支持通过 tf.Graph 函数来生成新的计算图，且不同计算图上的张量和运算不会共享，示例代码如下（代码位置：chapter02/new_graph.py）。

```
1  import tensorflow as tf
2  g1 = tf.Graph()
3  g2 = tf.Graph()
4  with g1.as_default():
5       v = tf.constant([1.0,2.0],name="v",dtype=tf.float32)
6  with g2.as_default():
7       v =tf.constant([3.0,4.0], name="v",dtype=tf.float32)
8  tensor1=g1.get_operation_by_name("v")
9  tensor2=g2.get_operation_by_name("v")
10 print(tensor1)
11 print(tensor2)
```

第 2 行代码：新建一个计算图 g1。

第 3 行代码：新建一个计算图 g2。

第 4～5 行代码：在计算图 g1 中定义张量 v。

第 6～7 行代码：在计算图 g2 中定义张量 v。

第 8～9 行代码：分别获得两个操作节点。
第 10～11 行代码：分别输出两个计算节点。
运行程序，输出结果如下。

```
name: "v"
op: "Const"
attr {
   key: "dtype"
   value {
     type: DT_FLOAT
   }
}
attr {
   key: "value"
   value {
     tensor {
       dtype: DT_FLOAT
       tensor_shape {
        dim {
          size: 2
        }
      }
      tensor_content: "\000\000\200?\000\000\000@"
    }
  }
}

name: "v"
op: "Const"
attr {
   key: "dtype"
   value {
     type: DT_FLOAT
   }
}
attr {
   key: "value"
   value {
     tensor {
       dtype: DT_FLOAT
       tensor_shape {
```

```
          dim {
            size: 2
          }
        }
        tensor_content: "\000\000@@\000\000\200@"
      }
    }
}
```

2.1.4　指定 GPU 设备

如果下载的 TensorFlow 是 GPU 版本，在程序运行过程中系统会自动检测可以利用的 GPU 来执行操作。

如果计算机上有不止一个 GPU 资源，除第一个之外，其他的 GPU 不参与计算。为了让 TensorFlow 能使用这些 GPU 资源，必须将节点明确地指派给定的 GPU 执行。with…device 语句用来指派给指定的 CPU 或 GPU。

以下代码通过 with…device 语句指定了 TesnorFlow 的运行设备（代码位置：chapter02/with_ device.py）。

```
1  import tensorflow as tf
2  with tf.device('/gpu:1'):
3       v1 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v1')
4       v2 = tf.constant([4.0, 5.0, 6.0], shape=[3], name='v2')
5       sum = v1 + v2
6  with tf.Session() as sess:
7       print (sess.run(sum))
```

第 2 行代码：指定在第 1 块 GPU 设备上运行。

在 TensorFlow 中，CPU 的名称为“/cpu:0”。如果有多个 CPU，则所有 CPU 都使用“/cpu:0”作为名称。GPU 不同，其名称为“/gpu:n”。n=1，表示第一个 GPU，以此类推。

第 3 行代码：tf.constant() 声明一个名为 v1 的浮点型张量，形状为 1 行 3 列，值是[1.0, 2.0,3.0]。

第 4 行代码：tf.constant() 声明一个名为 v2 的浮点型张量，形状为 1 行 3 列，值是[4.0, 5.0,6.0]。

第 5 行代码：v1 + v2 表示两个向量的加法算术运算，与 tf.add()函数的实现效果一致。

第 6 行代码：建立会话管理器，在会话中打印输出结果。其计算过程为[1.0+4.0, 2.0+5.0, 3.0+6.0]，最终的值为[5.0,7.0,9.0]。

类似地，还可以通过 tf.ConfigProto()来构建 config，在 config 中指定相关的 GPU，并

在会话中传入参数，绑定 GPU 进行操作。

tf.ConfigProto()函数相关参数的含义如表 2-1 所示。

表 2-1　tf.ConfigProto() 相关参数

参　数　名	含　　义
allow_soft_placement	如果指定 GPU 不存在，是否允许 TensorFlow 自动分配设备，可选值包括 True 和 False
log_device_placement	是否打印设备分类分配日志，可选值包括 True 和 False
gpu_options.allow_growth	是否允许 GPU 容量按需分配，即开始使用少量 GPU 资源，然后慢慢增加。可选值包括 True 和 False

以下代码演示了如何通过 ConfigProtol 类配置程序的运行设备（代码位置：chapter02/config_ proto.py）。

```
1  import tensorflow as tf
2  os.environ['CUDA_VISIBLE_DEVICES'] = '0,1'
3  config = tf.ConfigProto()
4  config.gpu_options.allow_growth = True
5  config.log_device_placement=True
6  v1 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v1')
7  v2 = tf.constant([1.0, 2.0, 3.0], shape=[3], name='v2')
8  sum = v1 + v2
9  with tf.Session(config = config) as sess:
10   print (sess.run(sum))
```

第 2 行代码：指定使用第 1、2 块 GPU。

第 3～5 行代码：建立 ConfigProto 对象，运行 GPU 资源按需分配，同时打印设备分类分配日志。

第 6～8 行代码：声明两个张量 v1 和 v2，并求两个张量的和。

第 9 行代码：在会话中传入 config 对象，从而利用 ConfigProto 对象的相关配置。

2.2　TensorFlow 数据模型

2.2.1　张量及属性

张量（Tensor）是 TensorFlow 中最重要的数据结构，用来表示程序中的数据。在计算图节点间传递的数据都是张量。可以把张量看作一个 n 维数组或列表，每个张量都包含类型（dtype）、阶（rank）与形状（shape）。

1．类型

TensorFlow 吸收了 Python 的原生数据类型，如布尔型，数值型（整数、浮点数）和字符串型。除此之外，TensorFlow 还有一些自有的数据类型。

TensorFlow 中张量的常见数据类型如表 2-2 所示。

表 2-2　张量类型

数 据 类 型	描　　述
tf.float32	32 位浮点数
tf.float64	64 位浮点数
tf.int64	64 位有符号整型
tf.int32	32 位有符号整型
tf.int16	16 位有符号整型
tf.int8	8 位有符号整型
tf.uint8	8 位无符号整型
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
tf.bool	布尔型
tf.complex64	由两个 32 位浮点数组成的复数：实数和虚数

以下代码定义了不同类型的张量（代码位置：chapter02/tensor_type.py）。

```
1  import tensorflow as tf
2  hello = tf.constant('Hello,world!', dtype=tf.string)
3  boolean = tf.constant(False, dtype=tf.bool)
4  int = tf.constant(100, dtype=tf.int8)
5  uint = tf.constant(100, dtype=tf.uint8)
6  float = tf.constant(100, dtype=tf.float32)
7  with tf.Session() as sess:
8     print(sess.run(hello))
9     print(sess.run(boolean))
10    print(sess.run(int))
11    print(sess.run(uint))
12    print(sess.run(float))
```

第 1 行代码：导入 tensorflow 类库，并简写为 tf。

第 2 行代码：定义张量 tf.string 是字符串类型。

第 3 行代码：定义张量 tf.bool 是布尔类型，如果改为‘false’就会报错，显示更改为字符串类型。

第 4 行代码：定义张量 tf.int8，为 8 位有符号整型张量。
第 5 行代码：定义张量 tf.uint8，为 8 位无符号整型张量。
第 6 行代码，tf.float32 定义 32 位浮点型张量。
第 7～12 行代码：在会话中输出各个张量。
运行程序，输出结果如下。

```
b'Hello,world!'
False
100
100
100.0
```

2．阶

在 TensorFlow 系统中，张量的维数被描述为阶。注意，张量的阶和矩阵的阶并不是同一个概念，张量的阶是张量维数的数量描述。例如，下面的张量是 2 阶张量。

t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

一阶张量可以认为是一个向量，二阶张量可以认为是二维数组，用 t[i, j] 访问其中的元素，三阶张量可以用 t[i, j, k] 访问其中的元素。

常见的张量及对应的阶如表 2-3 所示。

表 2-3　张量的阶

阶	实　例	例　子
0	纯量（只有大小）	a = 1
1	向量（大小和方向）	v = [1, 2, 3]
2	矩阵（二维数组）	m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3	3 阶张量（数据立体）	t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]
N	*n* 阶	*n* 中的括号数

3．形状

形状用于描述张量内部的组织关系。简单地讲，就是该张量有几行几列。

下面来定义一个张量，并分别输出张量的类型、形状以及阶，代码如下（代码位置：chapter02/tensor_property.py）。

```
1  import tensorflow as tf
2  c = tf.constant([[3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
3  print("张量类型：",c.dtype)
```

```
4  print("张量形状：",c.get_shape())
5  print("张量的阶：",c.get_shape().ndims)
```

第 2 行代码：定义一个 2 行 3 列的张量，第 1 行的值为[3.0, 4.0, 5.0]，第 2 行的值为[6.0, 7.0, 8.0]。

第 3 行代码：输出张量的数据类型。

第 4 行代码：输出张量的形状，即该张量有几行几列。

第 5 行代码：输出张量的阶。

运行代码，输出结果如下。

```
张量类型：  <dtype: 'float32'>
张量形状：  (2, 3)
张量的阶：  2
```

2.2.2　类型转换

TensorFlow 程序运行过程中，常常会涉及不同类型张量之间的相互转换。TensorFlow 提供了多种类型转换函数，其语法格式如表 2-4 所示。

表 2-4　TensorFlow 类型转换函数

函　数　名	含　　义
tf.string_to_number(string_tensor, out_type=None, name=None)	将字符串转换为数字。 **out_type**：可选 tf.float32、tf.int32，默认为 f.float32
tf.to_double(x, name=None)	将其他类型转换为 double 类型
tf.to_int32(x, name=None)	将其他类型转换为 int32
tf.cast(x, dtype, name=None)	将 x 的值转换为 dtype 指定的类型

下面的代码演示了如何将整型的张量转换为浮点型张量（代码位置：chapter02/tensor_ convert_type.py）。

```
1  import tensorflow as tf
2  float_tensor = tf.cast(tf.constant([1, 2, 3]), dtype=tf.float32)
3  init = tf.global_variables_initializer()
4  with tf.Session() as sess:
5       sess.run(init)
6       print(sess.run(float_tensor))
```

第 1 行代码：导入 tensorflow 类库，并简写为 tf。

第 2 行代码：tf.cast()函数将整型的张量转变为 32 位浮点型的张量。

第 3 行代码：tf.global_variables_initializer() 是一个全局初始化函数，在会话中运行，

用来初始化所有的节点。

第 4～5 行代码：创建会话，使用会话管理器管理会话。

第 6 行代码：在会话中输出张量的值，输出结果为[1. 2. 3.]，成功将整型转换为浮点型。

运行代码，输出结果如下。

```
[1. 2. 3.]
```

2.2.3　形状变换

在实际应用过程中，有时需要将张量从一个形状转换为另外一个形状，以满足计算需要。TensorFlow 中，reshape() 函数用来实现张量的形状变换，其语法格式如表 2-5 所示。

表 2-5　reshape()形状变换函数

函　　数	说　　明
tf.reshape(tensor, shape, name=None)	将张量从一个形状变换为另外一个形状。 **tensor**：待改变形状的 tensor 。 **shape**：必须是 int32 或 int64，决定了输出张量的形状。 **name**：可选，操作名称。 如果形状的一个分量是特殊值-1，则计算该维度的大小，以使总大小保持不变

如下代码演示了如何改变张量的形状（代码位置：chapter02/ reshape_tensor.py）。

```
1  import tensorflow as tf
2  c1 = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12], dtype=tf.float32,
   name="c1")
3  c2= tf.reshape(c1,(3,4))
4  c3 =tf.reshape(c1,( 2, -1, 3))
5  with tf.Session() as sess:
6       sess.run(tf.global_variables_initializer())
7       print(sess.run(c1))
8       print(sess.run(c2))
```

第 1 行代码：导入 tensorflow 类库，简写为 tf。

第 2 行代码：定义 c1 是一维张量，共有 12 个元素。

第 3 行代码：将 c1 转换成 3 行 4 列的元素，输出为：

```
[[ 1. 2. 3. 4.]
[ 5. 6. 7. 8.]
[ 9. 10. 11. 12.]]
```

第 4 行代码：将第 2 维度设置为-1，其维度根据第 1、3 维计算得出，输出为：

```
[[1. 2. 3. 4.]
 [ 5. 6. 7. 8.]
 [ 9. 10. 11. 12.]]
[[[ 1. 2. 3.]
  [ 4. 5. 6.]]
 [[ 7. 8. 9.]
  [10. 11. 12.]]]
```

2.3　变量的定义与使用

2.3.1　变量的定义与初始化

TensorFlow 中，变量（Variable）是一种特殊的张量（Tensor），其值可以是一个任意类型和形状的张量。与其他张量不同，变量存在于单个会话调用的上下文之外，主要作用是保存和更新模型中的参数。

声明变量通常使用 tf.Variable()函数，其语法格式如表 2-6 所示。

表 2-6　tf.Variable()函数

函　　数	说　　明
tf.Variable(initial_value, trainable=True, collections=None, validate_shape=True, name=None)	主要作用是声明一个变量。 initial_value：必选，指定变量的初始值。所有可转换为张量的类型均可。 trainable：可选，设置是否可以训练，默认为 True。 collections：可选，设置该图变量的类型，默认为 GraphKeys.GLOBAL_VARIABLES。 validate_shape：可选，默认为 True。如果为 False，则不进行类型和维度检查。 name：变量名称。如果未指定，系统会自动分配一个唯一的值

tf.Variable()的主要作用是构造一个变量并添加到计算图模型中。在运行其他操作之前，需要对所有变量进行初始化。最简单的初始化方法是添加一个对所有变量进行初始化的操作，然后在使用模型前运行此操作。最常见的方式是运行 tf.global_variables_initializer()函数进行全局初始化，该函数会初始化计算图中所有的变量。

下面的代码演示了如何在模型中初始化变量（代码位置：chapter02/variables_initializer.py）。

```
1  import tensorflow as tf
2  v = tf.Variable([1,2,3],dtype=tf.int32)
3  init_op = tf.global_variables_initializer()
4  with tf.Session() as sess:
5       sess.run(init_op)
6       print(sess.run(v))
```

第 2 行代码：tf.Variable() 定义了一个 1 行 3 列的整型变量，该变量的初始值为 1,2,3。

第 3 行代码：tf.global_variables_initializer() 定义了一个全局初始化操作。

第 5 行代码：在会话中运行 sess.run()，初始化模型中的所有变量。

运行代码，输出结果如下。

```
[1 2 3]
```

2.3.2 随机初始化变量

在声明变量时需要指定初始值，一般使用随机数给 Tensorflow 的变量初始化，常见的初始化方法如表 2-7 所示。

表 2-7　变量的初始化方法

函　　数	说　　明
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None)	产生一个符合正态分布的张量。 **shape**: 必选，生成张量的形状。 **mean**: 可选，正态分布的均值，默认为 0。 **stddev**: 可选，正态分布的标准差，默认为 1.0。 **dtype**: 可选，生成张量的类型，默认为 tf.float32。 **seed**: 可选，随机数种子，是一个整数。当设置之后，每次生成的随机数都一样
tf.truncated_normal(shape, mean=0, stddev=1.0)	产生一个满足正态分布的张量，当如果随机数偏离平均值超过 2 个标准差以上，将会被重新分配一个随机数。 **shape**: 必选，生成张量的形状。 **mean**: 可选，正态分布的均值，默认为 0。 **stddev**: 可选，正态分布的标准差，默认为 1.0
tf.random_uniform(shape, minval=low, maxval=high, dtype=tf.float32)	产生一个满足平均分布的张量。 **shape**: 必选，生成张量的形状。 **mean**: 必选，产生值的最小值。 **stddev**: 必选，产生值的最大值。 **dtype**：可选，产生值的类型，默认为 float32

下面的代码分别用不同的方式产生变量（代码位置：chapter02/parameter_initializer.py）。

```
1  import tensorflow as tf
2  w1 = tf.Variable(tf.random_normal([2, 3], stddev=1, seed=1))
3  w2= tf.truncated_normal(shape=[2,3], mean=0, stddev=1)
4  w3=tf.random_uniform((2,2), minval=1.0,maxval=2.0, dtype=tf.float32)
5  init_op = tf.global_variables_initializer()
6  with tf.Session() as sess:
7       sess.run(init_op)
8       print("w1:",sess.run(w1))
9       print("w2:",sess.run(w2))
10      print("w3:",sess.run(w3))
```

第 1 行代码：导入 tensorflow 类库，并简写为 tf。

第 2 行代码：产生一个符合正态分布的 2 行 3 列张量，均值为 0，方差为 1，随机种子为 1。

第 3 行代码：产生一个截断的 2 行 3 列张量，均值为 0，方差为 1。

第 4 行代码：产生一个符合均匀分布的 2 行 2 列张量，最小值为 1.0，最大值为 2.0。

第 7 行代码：在会话中初始化计算图中的所有变量。

第 8～10 行代码：在会话中输出各个参数的值。

运行程序，输出结果如下。

```
w1: [[-0.8113182   1.4845988   0.06532937]
 [-2.4427042   0.0992484   0.5912243 ]]
w2: [[-0.0075413  -0.6601458   0.01212148]
 [-0.49024445  1.4596753  -0.27039385]]
w3: [[1.9456019 1.7730622]
 [1.1844541 1.1060741]]
```

2.3.3　获取变量

除了可使用 tf.Variable()创建变量之外，还可以使用 tf.get_variable()函数创建或获取变量。tf. get_variable()函数用于创建变量时，它和 tf.Variable()的功能是等价的。

以下代码给出了通过两个函数创建变量的实例。

```
m = tf.Variable(tf.constant(1.0,shape=[1],name="m"))
n = tf.get_variable(shape=[1],name="n",initializer=tf.constant_initializer[1])
```

可以看出，tf.Variable()和 tf.get_variable()创建变量的过程是一样的。两者的最大区别

在于指定变量名称的参数不同。tf.Variable()函数中，变量名称是可选参数；tf.get_variable()函数中，变量名是必选参数，当变量名存在时，将直接获取变量。

tf.get_variable()函数的语法格式如表 2-8 所示。

表 2-8　tf.get_variable()函数

函　　数	说　　明
tf.get_variable(	用来初始化或获取变量。
name,	**name**：变量的名称，必填。
shape,	**shape**：变量的形状，必填。
initializer)	**initializer**：变量初始化的方法，选填

tf.get_variable()函数拥有一个变量检查机制，会检测已经存在的变量是否设置为共享变量。如果未设置为共享变量，TensorFlow 运行到第 2 个拥有相同名字变量的时候，就会抛出异常错误。

下面的代码描述了该种错误类型（代码位置：chapter02/ get_variable.py）。

```
1  import tensorflow as tf
2  a1= tf.get_variable(name='a', initializer=2)
3  a2 = tf.get_variable(name='a', initializer=2)
4  init_op = tf.global_variables_initializer()
5  with tf.Session() as sess:
6       sess.run(init_op)
7       print(sess.run(a1))
8       print(sess.run(a1))
```

第 2 行代码：定义了节点名称为 a 的变量 a1。

第 3 行代码：定义了节点名称为 a 的变量 a2。

运行程序，系统将发生崩溃，这表明 tf.get_variable()只能定义一次指定的变量名称，当第 3 行代码再将变量命名为 a 时，由于有同名的变量，因此程序发生了崩溃。

2.3.4　共享变量

tf.variable_scope()函数用来指定变量的作用域。不同作用域中的变量可以有相同的命名，包括使用 tf.get_variable()函数得到的变量以及 tf.Variable()函数创建的变量。

tf.get_variable()常常会配合 tf.variable_scope()一起使用，以实现变量共享。tf.variable_scope()函数会生成上下文管理器，并指定变量的作用域。

tf.variable_scope()里面还有一个 reuse=True 属性，表示使用已经定义过的变量，这时 tf.get_variable()不会创建新的变量，而是直接获取已经创建的变量。如果变量不存在，则会报错。

下面的代码使用了变量共享的功能（代码位置：chapter02/ variable_scope_reuse.py）。

```
1  import tensorflow as tf
2  with tf.variable_scope('V1'):
3       a1 = tf.get_variable(name='a1', shape=[1], initializer=tf.constant_
   initializer(1))
4  with tf.variable_scope('V2'):
5       a2 = tf.get_variable(name='a1', shape=[1], initializer=tf.constant_
   initializer(1))
6  with tf.variable_scope('V2',reuse=True) :
7       a3 = tf.get_variable('a1')
8  with tf.Session() as sess:
9       sess.run(tf.global_variables_initializer())
10      print(a1.name)
11      print(a2.name)
12      print(a3.name)
```

第 2～3 行代码：在 V1 变量空间中定义变量 a1。

第 4～5 行代码：在 V2 变量空间中定义变量 a1。由于两个 a1 位于不同的变量空间，所以不会产生冲突。

第 6～7 行代码：重用 V2 命名空间的 a1 变量。调用 tf.get_variable()时，会获取 V2 命名空间的 a1 变量的值。

运行代码，输出结果如下。

```
V1/a1:0
V2/a1:0
V2/a1:0
```

2.4　占位符与数据喂入机制

2.4.1　占位符定义

placeholder 是 TensorFlow 提供的占位符节点，由 tf.placeholder()函数创建，其实质上也是一种变量。占位符没有初始值，只会分配必要的内存，其值由会话中用户调用的 run()函数传递。占位符声明的方法如表 2-9 所示。

表 2-9　tf.placeholder()占位符形式

函　　数	说　　明
tf.placeholder(dtype, shape=None, name=None)	创建一个指定形状的占位符节点。 **dtype**：数据类型，必选，默认为 value 的数据类型。 **shape**：数据形状，必选，默认为 None，即一维值。也可以是多维，如[2,3], [None, 3]表示列为 3，行不定。 **name**：占位符名，可选，默认值不重复

2.4.2　数据喂入

TensorFlow 的数据供给机制允许在 TensorFlow 计算图中将数据注入任意张量中，然而却需要设置 placeholder 节点，通过 run()函数输入 feed_dict 参数，可以启动运算过程。

placeholder 节点被声明的时候是未被初始化的，也不包含任何数据，如果没有为它供给数据，则 TensorFlow 计算图运算的时候会产生错误。以下代码演示了如何向模型中喂入数据（代码位置：chapter02/feed_data.py）。

```
1  import tensorflow as tf
2  a = tf.placeholder(dtype=tf.int16)
3  b = tf.placeholder(dtype=tf.int16)
4  add = tf.add(a, b)          #a 与 b 相加
5  mul = tf.multiply(a, b)     #a 与 b 相乘
6  with tf.Session() as sess:
7       print ("相加: %i" % sess.run(add, feed_dict={a: 3, b: 4}))
8       print ("相乘: %i" % sess.run(mul, feed_dict={a: 3, b: 4}))
```

第 1 行代码：导入 tensorflow 类库，简写为 tf。

第 2～3 行代码：定义 a 和 b 两个整型占位符节点。

第 4～5 行代码：定义将 a 与 b 两个节点相加与相乘的运算节点。

第 7～8 行代码：在 session.run 中分别向相加、相乘运算喂入数据，并输出运算结果。

运行代码，运算结果如下。

```
相加: 7
相乘: 12
```

2.5　模型的保存与恢复

2.5.1　模型保存

训练完 TensorFlow 模型之后，可将其保存为文件，以便于预测新数据时直接加载使用。TensorFlow 模型主要包含网络的设计或者图以及已经训练好的网络参数的值。

TensorFlow 提供的 tf.train.Saver()函数可以建立一个 saver 对象，在会话中调用其 save()函数，即可将模型保存起来。代码如下。

```
import tensorflow as tf
saver = tf.train.Saver()
with tf.Session() as sess:
     sess.run(tf.global_variables_initializer())
     #将数据送入模型进行训练
     #训练完成后，使用 save()函数保存
     saver.save(sess,"savePath/filename")
```

save()函数的语法格式如表 2-10 所示。

表 2-10　save()函数

函　数	说　明
save(sess, save_path, global_step=None, latest_filename=None, meta_graph_suffix='meta', write_meta_graph=True, write_state=True)	**sess**：保存模型，要求必须有一个加载了计算图的会话，且所有变量已被初始化。 **save_path**：模型保存路径及保存名称。 **global_step**：如果提供，该数字会添加到 save_path 后，用于区分不同训练阶段的结果。 **latest_filename**：检查点文件的名字，默认是 checkpoint。 **meta_graph_suffix**：MetaGraphDef 元图后缀，默认为 meta。 **write_meta_graph**：是否要保存元图数据，默认为 True。 **write_state**：是否要保存 CheckpointStateProto，默认为 True

下面的程序展示了如何将两个张量的计算模型保存到本地计算机中（代码位置：chapter02/save_ model.py）。

```
1  import tensorflow as tf
2  m1 = tf.Variable(tf.constant(4.0, shape=[1]), name="m1")
3  m2 = tf.Variable(tf.constant(5.0, shape=[1]), name="m2")
4  result = m1 + m2
5  saver = tf.train.Saver()
6  with tf.Session() as sess:
7       sess.run(tf.global_variables_initializer())
8       saver.save(sess, "model/model.ckpt")
```

运行程序，当前目录的 model 文件夹下会产生 4 个文件：checkpoint、data-00000-of-00001、meta 和 index。下面介绍这 4 个文件分别保存的信息。

- checkpoint：保存模型的权重（weights）、偏置（biases）、梯度（gradients）以及

其他保存变量（variables）的二进制文件。

- data：保存模型的所有变量的值。
- meta：保存计算图的结构。当 meta 文件存在时，不在程序中定义模型，直接加载 meta 可以直接运行。
- index：保存 string-string 的键值对。其中的 key 值为张量名，value 为 BundleEntryProto。

2.5.2　模型恢复

模型保存好以后，载入非常方便。在会话中调用 saver 的 restore()函数，就会从指定的路径找到模型文件，并覆盖相关参数。saver.restore()函数的形式如表 2-11 所示。

表 2-11　模型恢复函数

函　　数	说　　明
saver.restore(sess, save_path)	从指定的路径恢复模型。 **sess**：用以恢复参数模型的会话。 **save_path**：已保存模型的路径，通常包含模型名字

加载模型时，需要定义 TensorFlow 计算图上的所有运算，但不需要进行变量的初始化，因为变量的值可以通过已经保存的模型加载进来。下面的代码演示了如何加载训练好的模型（代码位置：chapter02/restore.model.py）。

```
1  import tensorflow as tf
2  m1 = tf.Variable(tf.constant(7.0, shape=[1]), name="m1")
3  m2 = tf.Variable(tf.constant(8.0, shape=[1]), name="m2")
4  result = m1 + m2
5  saver = tf.train.Saver()
6  with tf.Session() as sess:
7       saver.restore(sess, "model/model.ckpt")
8       print(sess.run(result))
```

第 7 行代码：通过 saver.restore()将模型恢复到当前会话中。

第 8 行代码：输出模型的值。该值为 7，而不是 15。

有些时候，不希望重复定义图上的运算，也可以直接加载已经持久化的图，代码如下（代码位置：chapter02/restore.graph.py）。

```
1  import tensorflow as tf
2  saver=tf.train.import_meta_graph("model/model.ckpt.meta")
3  with tf.Session() as sess:
```

```
4       saver.restore(sess,'model/model.ckpt')
5       # 通过张量的名称来获取张量
6       print(sess.run(tf.get_default_graph().get_tensor_by_name('add:0')))
```

第 2 行代码：通过 tf.train.import_meta_graph()函数，直接加载持久化的图。

第 4 行代码：saver.restore()函数在当前会话中还原模型。

第 6 行代码：在会话中运行节点名称为 add 的张量，并输出计算图运算的结果。

运行代码，输出结果如下。

```
m1 [4.]
m2 [5.]
result [9.]
```

2.6　构建二维数据拟合模型

假设有一组数据集，满足对应关系 $y \approx 2x$。要求训练一个模型，输入 x，输出对应的 y。

2.6.1　准备数据

本例要求从-1 到 1 产生 100 个随机数据，代码如下（代码位置：chapter02/linear_model.py）。

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib.pyplot as plt
4  # (1)数据准备
5  x = np.linspace(-1, 1, 100)
6  y_ = 1 × x + np.random.randn(×x.shape) × 0.3
7  plt.plot(x, y_, 'ro', label='原始数据')
8  plt.show()
```

第 1～3 行代码：导入程序运行所需要的 tensorflow、numpy、matplotlib 库。

第 4 行代码：np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)，该函数的功能是在指定范围内按照固定的间隔生成数字。该代码在(−1,1)范围内产生 100 个数据。

第 5 行代码：np.random.randn()函数，生成随机的标准正态分布的数值。*是自动解包。

第 6～7 行代码：将产生的数据显示出来。

运行代码，程序结果如图 2-3 所示。

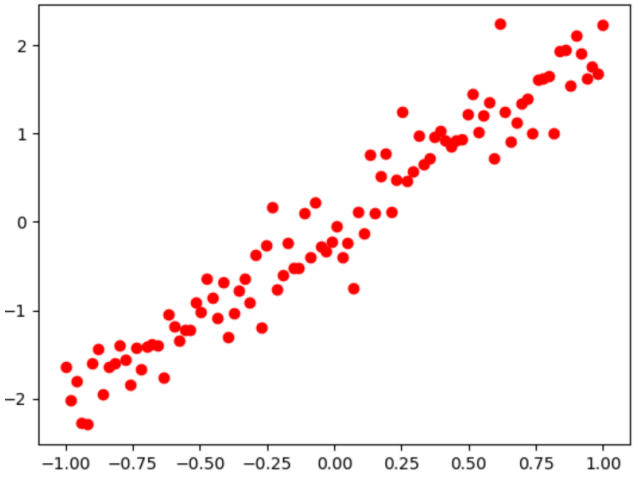


图 2-3　产生的随机数据集

2.6.2　搭建模型

模型的搭建过程实际上就是正向传播的过程，对应每一个输入，通过计算获得一个输出，其模型搭建过程代码如下（代码位置：chapter02/linear_model.py）。

```
   #(2)搭建模型
8  x=tf.placeholder(dtype=tf.float32)
9  y=tf.placeholder(dtype=tf.float32)
10 w=tf.Variable(tf.random_normal([1]),name='weight')
11 b=tf.Variable(tf.zeros([1]),name='bias')
12 z=tf.multiply(x,w)+b
```

第 8～9 行代码：x 和 y 代表占位符，使用 tf.placeholder()定义，一个代表 x 的输入，一个代表 y 的输出。

第 10～11 行代码：w 和 b 分别代表参数，w 被初始化为[−1,1]之间的参数，b 被初始化为全 0 的参数。

第 12 行代码：tf.multiply()函数的功能是两个变量相乘，再加上 b，就得到 z 的值。

2.6.3　反向传播

在模型训练过程中，得到 z 的值之后，该值与真实的 y 有差距，需要反向过程调整 w 和 b 的值，然后再次与真实的 y 比较，不断进行下去，直到将参数 w 和 b 的参数调整为合适的值为止。

反向传播的过程如下（代码位置：chapter02/linear_model.py）。

```
   #(3)反向传播
13 cost=tf.reduce_mean(tf.square(y-z))
14 learning_rate=0.05
15 optimizer=tf.train.GradientDescentOptimizer(learning_rate).minimize
   (cost)
```

第 13 行代码：tf.reduce_mean()函数用于生成真实值与预测值的平方差。

第 14 行代码：定义学习率 learning_rate 为 0.05，代表了参数的更新速度，这个值一般要小于 1。

第 15 行代码：tf.train.GradientDescentOptimizer()函数表示使用梯度下降法，最小化损失函数。

2.6.4　迭代训练

建立好模型后，就可以通过迭代来训练模型，TensorFlow 中的训练过程是通过会话来进行的。下面的代码首先进行全局化，然后设置训练迭代次数，启动会话进行训练，训练完成后保存模型（代码位置：chapter02/linear_model.py）。

```
   #(4)迭代训练
16 init_op=tf.global_variables_initializer()
17 training_epochs=100
18 display_step=10
19 saver = tf.train.Saver()
20 with tf.Session() as sess:
21      sess.run(init_op)
22      for epoch in range(training_epochs):
23           for (x_data,y_data) in zip(train_x,train_y):
24                sess.run(optimizer,feed_dict={x:x_data,y:y_data}
25                if epoch % display_step == 0:
26                     loss=sess.run(cost,feed_dict={x:x_data,y:y_data})
27                     print('Epoch:',epoch+1,'cost:',loss,'w:',sess.run(w),
   'b:',sess.run(b))
28           saver.save(sess,save_path="linear/linear.ckpt")
```

第 17 行代码：设置整个模型训练的轮数。

第 18 行代码：指定每 10 轮显示一次训练结果。

第 19 行代码：建立一个 saver 对象，用于保存模型。

第 22 行代码：for epoch in range()实现循环 100 轮。

第 23 行代码：从训练集中取出训练数据 x 和 y。
第 24 行代码：将训练数据 x 和 y 喂入神经网络。
第 25～27 行代码：迭代训练模型，每隔 10 轮打印一次 w 和 b 的值。
第 28 行代码：训练完成后，通过 saver.save 保存模型。
运行代码，其程序每轮训练的损失函数以及权重变化如下。

```
Epoch: 1 cost: 0.005744565 w: [0.5203005] b: [1.3085341]
Epoch: 11 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 21 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 31 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 41 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 51 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 61 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 71 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 81 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
Epoch: 91 cost: 0.0045678923 w: [1.9802843] b: [-0.00807061]
```

2.6.5　使用模型

模型训练完成并保存之后，就可以使用模型进行预测了，代码如下（代码位置：chapter02/ linear_model.py）。

```
   #(5)使用模型
29 with tf.Session() as sess:
30      saver.restore(sess, "linear/linear.ckpt")
31      print("模型的预测值为:",sess.run(z,feed_dict={x:0.6}))
```

第 30 行代码，通过 saver.restore()将训练好的模型加载到当前会话中。
第 31 行代码：通过向模型中喂入数据进行预测。
运行代码，输出结果如下。

```
模型的预测值为: [1.1801001]
```

2.7　本 章 小 结

TensorFlow 是一个基于计算图的深度学习编程模型，它的名字来源于本身的运行原理，Tensor 表示张量，其实质上是某种类型的多维数组，Flow 表示基于数据流图的计算。搭建基于 TensorFlow 的深度学习模型分为两个阶段：图的构建阶段和执行阶段。

图的构建阶段也称为图的定义阶段，该过程会在图模型中定义所需的运算，每次运

算的结果以及原始的输入数据都可称为一个节点。图的执行阶段是在会话中进行的，在会话中执行相关计算图中的节点，从而产生运算结果。

所有常量、变量、占位符都可以称为张量，每个张量都包含类型、阶和形状 3 个概念，在程序运行过程中，张量往往需要转换成另外一种形状或类型的张量，TensorFlow 提供了形状和类型转换的相关函数。

模型训练完成后，可以将模型文件保存在本地文件中，TensorFlow 提供了模型的保存和恢复操作，方便计算场景的迁移。

2.8 本章习题

1．选择题

（1）TensorFlow 是一个基于（　　）的编程模型。

A．编辑和解释程序　　B．面向过程

C．计算图　　D．面向对象

（2）TensorFlow 程序分为计算图构建阶段和执行阶段，第 1 阶段的主要任务是（　　）。

A．构建模型的计算节点　　B．定义数据模型

C．定义会话　　D．定义程序输入/输出

（3）运行 session.run(op)的含义是（　　）。

A．运行该行代码　　B．建立会话

C．在会话中运行计算图中名为 op 的节点　　D．以上说法都不对

（4）（　　）输出张量的类型（　　）。

A．dtype　　B．cdtype　　C．ndims　　D．以上都不是

（5）已知张量[[1, 2, 3],[4, 5, 6], [7, 8, 9]]，该张量的阶为（　　）。

A．1　　B．2　　C．3　　D．4

（6）tf.get_variable(shape=[1],name="m",initializer=tf.constant_initializer[1]) 关于上述代码，下面说法中正确的是（　　）。

A．创建一个名为 m 的变量，该变量的形状为一个元素，初始值为 0

B．创建一个名为 m 的变量，如果变量不存在则创建一个，如果存在则直接使用

C．获得一个名为 m 的变量

D．用值 1 初始化变量 m

（7）以下（　　）函数可以产生符合平均分布的变量。

A．tf.random_normal()　　B．tf.truncated_normal()

C．tf.random_uniform()　　D．tf.zeros_initializer()

（8）以下（　　）函数可以产生符合正太分布的变量。

A．tf.random_normal()　　B．tf.truncated_normal()

C．tf.random_uniform()　　D．tf.zeros_initializer()

（9）tf.get_variable 的（　　）机制会检测已经存在的变量是否设置为共享变量，如果遇到第 2 个拥有相同名字的变量的时候就会报错。

A．变量检测机制　　B．共享机制

C．变量获取机制　　D．以上都不对

（10）将数据喂入神经网络的关键字是（　　）。

A．feed　　B．fetch

C．feed_dict　　D．fetch_dict

2．填空题

（1）在 TensorFlow 中，程序的执行分为两个阶段，分别是_____和_____。

（2）TensorFlow 是一个基于_____的编程模型。

（3）有张量 m=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]，则该张量的维数是_____。

（4）声明一个张量 b = tf.constant([1,2,3],[4,5,6],dtype=tf.int32),则该张量的形状为_____。

（5）声明张量 a=tf.constant([[[1,2,3]]],dtype=tf.float32),则该张量的阶是_____。

（6）在 TensorFlow 中实现模型保存的函数是_____。

（7）获得计算图中的节点的函数是_____。

3．判断题

（1）TensorFlow 的 Tensor 表示张量，而 Flow 表示张量在不同及节点之间的流动。（　　）

（2）计算图只有在会话提供的环境中才能运行，分配计算资源。（　　）

（3）在 TensorFlow 中，若声明字符串类型，可以直接使用 tf.string。（　　）

（4）在 TensorFlow 中，reshape()可以实现张量不同形状的变换。（　　）

（5）tf.zeros_initializer()用来初始化全 0 的张量。（　　）

（6）在 TensorFlow 中，tf.name_scope()主要实现共享变量的目的。 （ ）

4．简答题

（1）简述 TensorFlow 模型的运行机制。

（2）简述张量在模型运行中的作用。

（3）常量、变量和占位符之间有什么区别与联系？

（4）数据喂入机制的作用是什么？

（5）如何保存与恢复模型？

5．编程题

（1）已知两个张量 [1,3,5,7] 和 [2,4,4,8]，编写一个模型，计算两个张量的加法，并输出结果。

（2）已知张量 [1,2,3,4,5,6,7,8,9,10,11,12] 有 12 个元素，利用 tf.reshape()将其形状转换为 [2,3,2] 的三维张量。

（3）已知 $y \approx x^3$，编写一个模型，拟合该函数。