第 1 章 TensorFlow与编程

CHAPTER 1

TensorFlow 是一个采用数据流图（data flow graphs），用于数值计算的开源软件库。其命名来源于本身的原理，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 的运行过程就是张量从图的一端流动到另一端的计算过程。张量从图中流过的直观图像是其取名为 TensorFlow 的原因。

1.1 语言与系统的支持

TensorFlow 支持多种客户端语言下的安装和运行，目前最新版本为 2.12.1，新版本提供了更多的 Bug 修复和功能改进，还针对漏洞发布了补丁。

1. Python

TensorFlow 提供 Python 语言下的四个不同版本：CPU 版本（TensorFlow）、包含 GPU 加速的版本（TensorFlow-gpu），以及两个编译版本（tf-nightly、tf-nightly-gpu）。TensorFlow 的 Python 版本支持 Ubuntu16.04、Windows 7、macOS10.12.6 Sierra、Raspbian 9.0 及对应的更高版本，其中 macOS 版不包含 GPU 加速。安装 Python 版 TensorFlow 可以使用模块管理工具 pip/pip3 或 anaconda 并在终端直接运行。

此外 Python 版 TensorFlow 也可以使用 Docker 安装。

2. C

TensorFlow 提供 C 语言下的 API，可以用于构建其他语言的 API，支持 x86-64 下的 Linux 类系统和 macOS 10.12.6 Sierra 或其更高版本，macOS 版不包含 GPU 加速。其安装过程如下：

- 下载 TensorFlow 预编译的 C 文件到本地系统路径（通常为 usr/local/lib）并解压缩。
- 使用 ldconfig 编译链接。

用户也可在其他路径解压文件并手动编译链接，此外，编译 C 接口时需确保本地的 C 编译器（例如 gcc）能够访问 TensorFlow 库。

3. 配置 GPU

TensorFlow 支持在 Linux 和 Window 系统下使用统一计算架构（Compute Unified Device Architecture，CUDA）高于 3.5 的 NVIDIAGPU。配置 GPU 时要求系统有 NVIDIA GPU 驱动 384.x 及以上版本、CUDA Toolkit 和 CUPTI（CUDA Profiling Tools Interface）9.0 版本、cuDNN SDK7.2 以上版本。其可选配置包括 NCCL 2.2（用于多 GPU 支持）、TensorRT 4.0（用于 TensorFlow 模型优化）。

1.2 TensFlow 的特点

TensorFlow 到底有什么特点，能让它在这么短的时间内得到如此广泛的应用呢？下面就简要介绍 TensorFlow 的特点。

1. 高度灵活性

TensorFlow 不仅是一个深度学习库，所有可以把计算过程表示成一个数据流图的过程，都可以用它来计算。TensorFlow 允许用计算图的方式建立计算网络，同时又可以很方便地对网络进行操作。

2. 真正的可移植性

TensorFlow 可以在台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等上运行。

3. 自动求微分

TensorFlow 自动求微分的能力非常适用于基于梯度的机器学习算法。作为 TensorFlow 用户，只需要定义预测模型的结构，将这个结构和目标函数（objective function）结合在一起，并添加数据，就可以用 TensorFlow 自动计算相关的微分导数。

4. 多语言支持

TensorFlow 采用非常易用的 Python 来构建和执行计算图，同时也支持 C++、Java、Go 语言。

5. 丰富的算法库

TensorFlow 提供了所有开源机器学习框架里最全的算法库，并且还在不断添加新的算法库。这些算法库已经能满足大部分需求，对于普通应用，基本上不用再去自定义算法库。

6. 大量的开源项目

TensorFlow 在 GitHub 上的主项目下还有类似 models 这样的项目，里面包含了许多应用领域的最新研究算法的代码实现，比如图像识别领域效果最好的 Inception 网络和残差网络，能够让机器自动用文字描述一张图片的 im2txt 项目，自然语言某些处理领域达到人类专家水平的 syntaxnet 项目，等等。

7. 性能最优化

假设有一个 32 个 CPU 内核、4 个 GPU 显卡的工作站，想要将工作站的计算潜能全部发挥出来。TensorFlow 由于给予了线程、队列、异步操作等最佳的支持，故可将硬件的计算潜能全部发挥出来。

8. 将科研和产品联系在一起

目前谷歌的科学家已经用 TensorFlow 尝试新的算法，产品团队采用 TensorFlow 来训练和使用计算模型，并直接提供给在线用户。使用 TensorFlow 可以让应用型研究者将想法迅速运用到产品中，也可以让学术性研究者更直接地彼此分享代码，从而提高科研产出率。

1.3　TensorFlow 的环境搭建

本节主要介绍如何在几个主要的平台上安装 TensorFlow，以及对其进行简单的运行测试。

1.3.1　安装环境介绍

目前 TensorFlow 社区推荐的环境是 Ubuntu，但是 TensorFlow 同时支持 macOS 及 Windows 上的安装部署。

在深度学习计算过程中，大量的操作是向量和矩阵的计算，而 GPU 在向量和矩阵计算方面比 CPU 有一个数量级的速度提升，显然机器学习在 GPU 上运算效率更高，所以推荐在 GPU 的机器上运行 TensorFlow 程序。

1. CUDA 简介

显卡厂商 NVIDIA 推出的运算平台 CUDA（Compute Unified Device Architecture），是一种通用的并行计算架构，该架构使 GPU 能够解决复杂的计算问题，它包含了 CUDA 指令集以及 GPU 内部的并行计算引擎。它还提供了硬件的直接访问接口，因此不必像传统方式一样必须依赖图形 API 接口来实现 GPU 的访问，从而给大规模的数据计算应用提供

了一种比 GPU 更加强大的计算能力。程序开发人员通过 C 语言，利用 CUDA 的指令接口，就可以编写 CUDA 架构的程序。

2. CuDNN 简介

CuDNN 的全称是 CUDA Deep Neural Network library，是专门针对深度学习框架设计的一套 GPU 计算加速方案，最新版本的 CuDNN 提供了对深度神经网络中的向前向后的卷积、池化以及 RNN 的性能优化。目前，大部分深度学习框架都支持 CuDNN。

目前包括 TensorFlow 在内的大部分深度学习框架都支持 CUDA，所以为了让深度神经网络的程序在 TensorFlow 上运行得更好，推荐至少配置一块支持 CUDA 和 CuDNN 的 NVIDIA 的显卡。

3. 查看机器的显卡信息

下面从 Windows 系统上查看机器的显卡信息。

在“运行”对话框中输入 dxdiag，如图 1-1 所示，然后单击“确定”按钮，此时会打开“DirectX 诊断工具”窗口。单击其中的“显示”标签页，就可以查看机器的显卡信息，如图 1-2 所示。

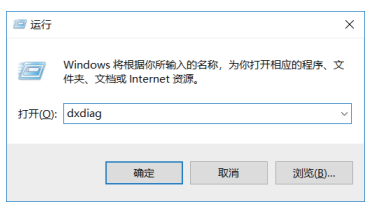

图 1-1　输入 dxdiag 命令

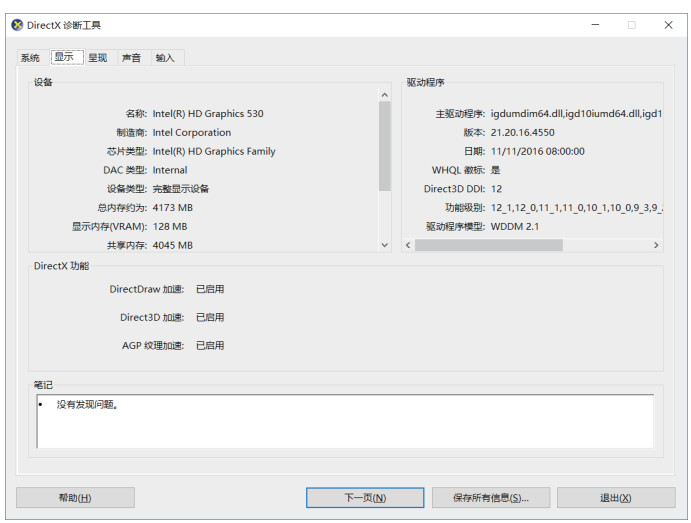

图 1-2　查看机器的显卡信息

从图 1-2 中可以看到，这个机器上的显卡是 Intel(R) HD Graphics Family。

1.3.2　安装 TensorFlow

TensorFlow 的 Python 语言 API 支持 Python 2.7 和 Python 3.3 以上的版本。GPU 版本推荐使用 CUDA Toolkit 8.0 和 CuDNN v5. 版本，CUDA 和 CuDNN 的其他版本也支持，不过需采用自己编译源码的方式安装。

1. 安装 pip

pip 是用来安装和管理 Python 包的管理工具，在此首先介绍它在各个平台上的安装方法。在安装 pip 之前，请先自行安装好 Python。

1）Ubuntu/Linux

在 Ubuntu/Linux 系统上安装 pip 的命令如下：

```
$ sudo apt-get install python3-pip
```

2）macOS

在 macOS 系统上安装 pip 的命令如下：

```
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```

3）Windows

在 Windows 系统上安装 pip 的命令如下：

```
# 去 Python 官网下载 pip
https://pypi.python.org/pypi/pip#downloads
# 解压文件，通过命令行安装 pip
>> python setup.py install
# 设置环境变量
```

在 Windows 的环境变量的 PATH 变量的最后添加“\Python 安装目录 \Scripts”。

2. 通过 pip 安装 TensorFlow

TensorFlow 已经把最新版本的安装程序上传到了 PyPI，所以可以通过最简单的方式来安装 TensorFlow（要求 pip 版本在 8.1 版本或者更高）。

安装 CPU 版本的 TensorFlow 的命令如下：

```
sudo pip3 install TensorFlow
```

安装支持 GPU 版本的 TensorFlow 的命令如下：

```
sudo pip3 install TensorFlow-gpu
```

在 Windows 系统上安装 CPU 版本。

```
C:\> pip install --upgrade
```

3. 源码编译安装 TensorFlow

有时需要单独自定义一下 TensorFlow 的安装，比如 CUDA 是 7.5 版本的，可能需要自己编译源码进行安装。在此只介绍在 Ubuntu 系统上如何通过源码编译安装。

1）从 git 上下载源码

从 git 上下载源码的地址为 git clone https://github.com/TensorFlow/TensorFlow。

2）安装 Bazel

Bazel 是谷歌开源的一套自动化构建工具，可以通过源的方式安装，也可以通过编译源码安装，这里只介绍通过源的方式安装。

首先，安装 JDK8。

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

接着，添加 Bazel 源的地址。

```
$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable
jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel.list
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
```

1.3.3 安装测试

到这里就成功安装好了 TensorFlow，下面简单测试一下安装是否成功。

```
>>> import TensorFlow as tf
>>> print(tf.__version__)
2.12.1
```

上面这段代码若正常运行会打印 TensorFlow 的版本号，这里是“2.12.1”。但经常会存在一些问题，例如，如果在 import TensorFlow as tf 之后打印没有显示 CUDA 或者 CUDNN，一般是因为 CUDA 或者 CUDNN 的路径没有添加到环境变量中。

再通过一个简单的计算查看 TensorFlow 是否运行正常。输入如下代码：

```
>>> import TensorFlow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess=tf.compat.v1.Session()
>>> sess
<TensorFlow.python.client.session.Session object at 0x00000216881BD310>

>>> print(sess.run(hello))
b'Hello, TensorFlow!'
>>> a = tf.constant(1)
>>> b = tf.constant(2)
>>> c=sess.run(a + b)
>>> print("1+2= %d" % c)
1+2=3
```

如果这段代码可以正常输出“Hello, TensorFlow!”和“1+2=3”，即说明 TensorFlow 已经安装成功。

1.4　张量

TensorFlow 是一个开源软件库，使用数据流图进行数值计算。图中的节点表示数学运算，而图边表示在它们之间传递的多维数据数组（张量）。

Tensor 的意思是“张量”，Flow 的意思是“流或流动”。任意维度的数据都可以称作张量，如一维数组、二维数组、*N* 维数组。

1.4.1　张量的概念

张量可以被简单理解为多维阵列，其中零阶张量表示标量，也就是一个数。一阶张量表示一维阵列；*n* 阶张量表示 *n* 维阵列。

张量在 TensorFlow 中的实现不是直接采用阵列的形式，它只是对 TensorFlow 中运算结果的引用。张量中并没有真正存储数字，它存储的是得到这些数字的计算过程。例如：

```
import TensorFlow as tf
a = tf.constant([1.0, 2.0], name="a")
b = tf.constant([2.0, 3.0], name="b")
res = tf.add(a, b, name="add")
print (res)
```

运行程序，输出如下：

```
Tensor("add:0", shape=(2,), dtype=float32)
```

TensorFlow 中的张量和 Numpy 中的阵列不同，TensorFlow 的计算结果不是一个具体的数字，而是一个张量的结构。一个张量存储了三个属性：名字、维度和型别。表 1-1 列出了张量的数据类型。

表 1–1　张量的数据类型

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_INT8	tf.int8	8 位有符号整型
DT_INT16	tf.int16	16 位有符号整型
DT_INT32	tf.int32	32 位有符号整型
DT_INT64	tf.int64	64 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

1.4.2　张量的使用

张量的使用可以总结为两大类。

第一类使用情况是对中间计算结果的引用。当一个计算包含很多计算结果时，使用张量可以提高代码的可读性。下面代码对使用张量和不使用张量记录中间结果来完成向量相加进行了对比。

```
import TensorFlow as tf
# 使用张量记录中间结果
a=tf.constant([1.0,2.0],name='a')
b=tf.constant([2.0,3.0],name='b')
result=a+b

# 直接结算
result=tf.constant([1.0,2.0],name='a')+
tf.constant([2.0,3.0],name='b')
```

从上面的程序样例可以看到 a 和 b 其实就是对常量生成这个运算结果的引用，这样在做加法时可以直接使用这两个变量，而不需要再去生成这些常量。同时通过张量来存储中间结果，可以很方便地获取中间结果。比如在卷积神经网络中，卷积层或池化层有可能改变张量的维度，通过 result.get_shape 函数来获取结果张量的维度信息可以免去人工计算的麻烦。

第二类使用情况是当计算图构造完成之后，可以用张量来获取计算结果，即得到真实的数字。虽然张量本身没有存储具体的数字，但可以通过会话 session 得到这些具体的数字，比如使用 tf.Session().run(result) 语句得到计算结果。

1.4.3　NumPy 库

TensorFlow 的数据类型是基于 NumPy 的数据类型。例如：

```
>>> import numpy as np
>>> import TensorFlow as tf
>>> np.int64 == tf.int64
True
>>>
```

任何一个 NumPy 数组均可传递给 TensorFlow 对象。

对于数值类型和布尔类型而言，TensorFlow 和 NumPy dtype 的属性是完全一致的，但在 NumPy 中并无与 tf.string 精确对应的类型。TensorFlow 可以从 NumPy 中导入字符串数组，只是不要在 NumPy 中显式指定 dtype。

在运行数据流图之前和之后，都可以利用 NumPy 库的功能，因为从 Session.run 方法返回的张量均为 NumPy 数组。例如：

```
>>> t1 = np.array(50, dtype= np.int64)
# 在 NumPy 中使用字符串时，不要显式指定 dtype 属性
>>> t2 = np.array([b'apple', b'peach', b'grape'])

>>> t3 = np.array([[True, False, False],[False, False, True],[False, True,
True]], dtype = np.bool)
>>> t4 = np.array([[[1]]], dtype= np.float32)
```

TensorFlow 是为理解 NumPy 原生数据类型而设计的，但反过来行不通，所以不要尝试用 tf.int32 去初始化一个 NumPy 数组。

```
>>> a = np.array(3, np.int) # 不是 np.int32
>>> a.dtype
dtype('int32')
>>>
>>> a = np.array(3, np.float)
>>> a.dtype
dtype('float64')
>>>
>>> a.dtype
dtype('float64')
>>>
>>> a = np.array(3, np.float32)
>>> a.dtype
dtype('float32')
```

```
>>>
```

手工指定 Tensor 对象时，使用 NumPy 是推荐的方式。

1.4.4 张量的阶

张量的阶（rank）表征了张量的维度，但是与矩阵的秩（rank）不一样，它的阶表示张量的维度的质量。

阶为 1 的张量等价于向量，阶为 2 的张量等价于矩阵。对于一个阶为 2 的张量，通过 $t[i, j]$ 就能获取它的每个元素。对于一个阶为 3 的张量，需要通过 $t[i, j, k]$ 进行寻址，以此类推，如表 1-2 所示。

表 1–2 张量的阶

阶	数学实体	实 例
0	Scalar	scalar=999
1	Vector	vector=[3,6,9]
2	Matrix	matrix=[[1,4,7],[2,5,8],[3,6,9]]
3	3-tensor	tensor=[[[5],[1],[3]],[[99],[9],[100]],[[0],[8],[2]]]
n	n-tensor	...

在下面的例子中，可创建一个张量，获取其结果。

```
>>> import TensorFlow as tf
>>> tens1=tf.constant([[[1,2],[3,6]],[[-1,7],[9,12]]])
>>> sess=tf.Session()
```

这个张量的阶是 3，因为该张量包含的矩阵中的每个元素都是一个向量。

1.4.5 张量的形状

TensorFlow 使用三个术语描述张量的维度：阶（rank）、形状（shape）和维数（dimension number）。三者之间的关系如表 1-3 所示。

表 1–3 三者之间的关系

阶	形 状	维 数	实 例
0	[]	0-D	5
1	[D0]	1-D	[4]
2	[D0,D1]	2-D	[3,9]
3	[D0,D1,D2]	3-D	[1,4,0]
n	[D0,D1,...,D*n*−1]	*n*−D	形为 [D0,D1,...,D*n*−1] 的张量

如下代码创建了一个三阶张量，并打印出它的形状。

```
>>> import TensorFlow as tf
>>> tens1=tf.constant([[[1,2],[3,6]],[[-1,7],[9,12]]])
>>> tens1
<tf.Tensor 'Const:0' shape=(2, 2, 2) dtype=int32>
>>> printf sess.run(tens1) [1,1,0]
```

1.5　认识变量

从初识 tf 开始，变量这个名词就很重要，因为深度模型往往所要获得的就是通过参数和函数对某一或某些具体事物的抽象表达。而那些未知的数据需要通过学习而获得，在学习的过程中它们不断变化着，最终收敛达到较好的表达能力，因此它们无疑是变量。

训练模型时，用变量来存储和更新参数。变量包含张量（tensor）存放于内存的缓存区。建模时它们需要被明确地初始化，模型训练后它们必须被存储到磁盘。这些变量的值可在之后模型训练和分析时被加载。

通过之前的学习，可以列举出以下 tf 的函数：

```
var = tf.get_variable(name, shape, initializer=initializer)
global_step = tf.Variable(0, trainable=False)
init = tf.initialize_all_variables()# 高版本 tf 已经舍弃该函数，改用 global_
                                    # variables_initializer()
saver = tf.train.Saver(tf.global_variables())
initial = tf.constant(0.1, shape=shape)
initial = tf.truncated_normal(shape, stddev=0.1)
tf.global_variables_initializer()
```

上述函数都和 tf 的参数有关，主要包含在以下两类中：

（1）tf.Variable 类；

（2）tf.train.Saver 类。

从变量存在的整个过程来看主要包括变量的创建、初始化、更新、保存和加载。

1.5.1　变量的创建

创建一个变量时，会将一个张量作为初始值传入构造函数 variable()。tf 提供了一系列操作符来初始化张量，初始值是常量或随机值。注意，所有这些操作符都需要指定张量的 shape。变量的 shape 通常是固定的，但 TensorFlow 提供了高级的机制来重新调整其行列数。

可以创建以下类型的变量：常数、序列、随机数。

【例 1-1】创建常数变量。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()

# 常数 constant
tensor=tf.constant([[1,3,5],[8,0,7]])
# 创建 tensor 值为 0 的变量
x = tf.zeros([3,4])
# 创建 tensor 值为 1 的变量
x1 = tf.ones([3,4])
# 创建 shape 和 tensor 一样，但值全为 0 的变量
y = tf.zeros_like(tensor)
# 创建 shape 和 tensor 一样，但值全为 1 的变量
y1 = tf.ones_like(tensor)
# 用 8 填充 shape 为 2*3 的 tensor 变量
z = tf.fill([2,3],8)
sess = tf.compat.v1.Session()
sess.run(tf.compat.v1.global_variables_initializer())
print (sess.run(x))
print (sess.run(y))
print (sess.run(tensor))
print (sess.run(x1))
print (sess.run(y1))
print (sess.run(z))
```

运行程序，输出如下：

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0 0 0]
 [0 0 0]]
[[1 3 5]
 [8 0 7]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[1 1 1]
 [1 1 1]]
[[8 8 8]
 [8 8 8]]
```

【例 1-2】创建数字序列变量。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
x=tf.linspace(10.0, 15.0, 3, name="linspace")
y=tf.compat.v1.lin_space(10.0, 15.0, 3)
w=tf.range(8.0, 13.0, 2.0)
z=tf.range(3, -3, -2)
sess = tf.compat.v1.Session()
sess.run(tf.compat.v1.global_variables_initializer())
print (sess.run(x))
print (sess.run(y))
print (sess.run(w))
print (sess.run(z))
```

运行程序，输出如下：

```
[10.  12.5 15. ]
[10.  12.5 15. ]
[ 8. 10. 12.]
[ 3  1 -1]
```

此外，TensorFlow 有几个操作用来创建不同分布的随机张量。注意随机操作是有状态的，并在每次评估时创建新的随机值。

下面是一些与随机张量相关的函数的介绍。

（1）tf.random_normal() 函数。

该函数用于从正态分布中输出随机值，其语法格式为：

```
random_normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.float32,
    seed=None,
    name=None
)
```

其中，参数含义如下。

- shape：一维整数或 TensorFlow 数组表示输出张量的形状。
- mean：dtype 类型的 0-D 张量或 TensorFlow 值表示正态分布的均值。
- stddev：dtype 类型的 0-D 张量或 TensorFlow 值表示正态分布的标准差。

- dtype：输出的类型。
- seed：一个 TensorFlow 整数，用于为分发创建一个随机种子。
- name：操作的名称（可选）。
- 返回：将返回一个指定形状的张量，采用符合要求的随机值填充。

（2）tf.truncated_normal() 函数。

该函数生成的值遵循具有指定平均值和标准差的正态分布，和 tf.random_normal( ) 函数的不同之处在于，其平均值大于 2 个标准差的值将被丢弃并重新选择。其语法格式为：

```
tf.truncated_normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.float32,
    seed=None,
    name=None
)
```

其中，参数含义如下。

- shape：一维整数或 TensorFlow 数组表示输出张量的形状。
- mean：dtype 类型的 0-D 张量或 TensorFlow 值表示截断正态分布的均值。
- stddev：dtype 类型的 0-D 张量或 TensorFlow 值表示截断前正态分布的标准偏差。
- dtype：输出的类型。
- seed：一个 TensorFlow 整数，用于为分发创建随机种子。
- name：操作的名称（可选）。
- 返回：函数返回指定形状的张量，采用随机截断的符合要求的值填充。

（3）tf.random_uniform() 函数。

该函数从均匀分布中输出随机值，其语法格式为：

```
random_uniform(
    shape,
    minval=0,
    maxval=None,
    dtype=tf.float32,
    seed=None,
    name=None
)
```

其中，生成的值在 [minval, maxval) 范围内遵循均匀分布，下限 minval 包含在范围内，

而上限 maxval 被排除在外。其参数含义如下。

- shape：一维整数或 TensorFlow 数组表示输出张量的形状。
- minval：dtype 类型的 0-D 张量或 TensorFlow 值，要生成的随机值范围的下限，默认为 0。
- maxval：dtype 类型的 0-D 张量或 TensorFlow 值，要生成的随机值范围的上限，如果 dtype 是浮点，则默认为 1。
- dtype：输出的类型，如 float16、float32、float64、int32、orint64。
- seed：一个 TensorFlow 整数，用于为分布创建一个随机种子。
- name：操作的名称（可选）。
- 返回：返回填充随机均匀值的指定形状的张量。

（4）tf.random_shuffle() 函数。

该函数用于随机地将张量沿其第一维度打乱，其语法格式为：

```
random_shuffle(
    value,
    seed=None,
    name=None
)
```

张量沿着维度 0 被重新打乱，使得每个 value[i][j] 被映射到唯一一个 output[m][j]。例如，一个 3×2 张量可能出现的映射为：

```
[[1, 2],       [[5, 6],
 [3, 4],  ==>   [1, 2],
 [5, 6]]        [3, 4]]
```

其参数含义如下。

- value：将被打乱的张量。
- seed：一个 TensorFlow 整数，用于为分布创建一个随机种子。
- name：操作的名称（可选）。
- 返回：与 value 具有相同的形状和类型的张量，沿着它的第一个维度打乱。

（5）tf.random_crop() 函数。

该函数用于随机地将张量裁剪为给定的大小，其语法格式为：

```
random_crop(
    value,
    size,
```

```
        seed=None,
        name=None
    )
```

以一致选择的偏移量将一个形状 size 部分从 value 中切出，需要满足的条件为 value.shape >= size。

如果大小不能裁剪，会传递该维度的完整大小。例如，可以使用 size = [crop_height, crop_width, 3] 裁剪 RGB 图像。

cifar10 中就有利用该函数随机裁剪 24 × 24 大小的彩色图片的例子，代码如下：

```
distorted_image = tf.random_crop(reshaped_image, [height, width, 3])
```

random_crop( ) 函数的参数含义如下。

- value：向裁剪输入张量。
- size：一维张量，大小等级为 value。
- seed：TensorFlow 整数，用于创建一个随机的种子。
- name：此操作的名称（可选）。
- 返回：与 value 具有相同的秩并且与 size 具有相同形状的裁剪张量。

（6）tf.multinomial() 函数。

该函数用于从多项式分布中抽取样本，其语法格式为：

```
multinomial(
    logits,
    num_samples,
    seed=None,
    name=None
)
```

其中，参数含义如下。

- logits：形状为 [batch_size, num_classes] 的二维张量；每个切片 [i, :] 都表示所有类的非标准化对数概率。
- num_samples：0 维张量，是为每行切片绘制的独立样本数。
- seed：TensorFlow 整数，用于为分布创建一个随机种子。
- name：操作的名称（可选）。
- 返回：返回绘制样品的形状 [batch_size, num_samples]。

（7）tf.random_gamma() 函数。

该函数用于从每个给定的伽马分布中绘制 shape 样本，其语法格式为：

```
random_gamma(
    shape,
    alpha,
    beta=None,
    dtype=tf.float32,
    seed=None,
    name=None
)
```

其中，alpha 是形状参数，beta 是尺度参数。其他参数含义如下。

- shape：一维整数张量或 TensorFlow 数组。输出样本的形状是按照 alpha/beta-parameterized 分布绘制的。
- alpha：一个张量或者 TensorFlow 值或者 dtype 类型的 *n*-D 数组。
- beta：一个张量或者 TensorFlow 值或者 dtype 类型的 *n*-D 数组，默认为 1。
- dtype：alpha、beta 的类型，输出 float16、float32 或 float64。
- seed：一个 TensorFlow 整数，用于为分布创建一个随机种子。
- name：操作的名称（可选）。
- 返回 samples：具有 dtype 类型值的带有形状 tf.concat(shape, tf.shape(alpha + beta)) 的 Tensor。

（8）tf.set_random_seed() 函数。

该函数用于设置图形级随机 seed，作用在于可以在不同的图中重复那些随机变量的值，其语法格式为：

```
set_random_seed(seed)
```

可以从两个 seed 中获得依赖随机 seed 的操作：图层级 seed 和操作级 seed。seed 必须是整数，对大小没有要求，只是作为图层级和操作级标记使用。下面介绍如何设置 seed。

图层级 seed 与操作级 seed 的关系如下。

- 如果既没有设置图层级也没有设置操作级的 seed，则使用随机 seed 进行该操作。
- 如果设置了图层级 seed，但没有设置操作级 seed，则系统确定性地选择与图层级 seed 结合的操作级 seed，以便获得唯一的随机序列。
- 如果未设置图层级 seed，但设置了操作级 seed，则使用默认的图层级 seed 和指定的操作级 seed 来确定随机序列。
- 如果图层级 seed 和操作级 seed 都被设置了，则两个 seed 将一起用于确定随机序列。

具体来说，使用 seed，牢记以下 3 点。

① 要在会话的不同图中生成不同的序列，不要设置图层级 seed 或操作级 seed。

② 要为会话中的操作在不同图中生成相同的可重复序列，设置操作级 seed。

③ 要使所有操作生成的随机序列在会话中的不同图中都可重复，设置图层级 seed。

【例 1-3】创建随机变量。

```
# 不同情况请注释或取消注释相关语句
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
# 第一种情形：无 seed
a = tf.compat.v1.random_uniform([1])
# 第二种情形：操作级 seed
#a = tf.random_uniform([1], seed=-8)
# 第三种情形：图层级 seed
#tf.set_random_seed(1234)
#a = tf.random_uniform([1])
b = tf.compat.v1.random_normal([1])
tf.compat.v1.global_variables_initializer()

print("Session 1")
with tf.compat.v1.Session() as sess1:
  print(sess1.run(a))  # a1
  print(sess1.run(a))  # a2
  print(sess1.run(b))  # b1
  print(sess1.run(b))  # b2
print("Session 2")
with tf.compat.v1.Session() as sess2:
  print(sess2.run(a))  # a3(第一种情形 a1!=a3; 第二种情形 a1==a3; 第三种情形 a1==a3)
  print(sess2.run(a))  # a4(同上)
  print(sess2.run(b))  # b3(第一种情形 b1!=b3; 第二种情形 b1!=b3; 第三种情形 b1==b3)
  print(sess2.run(b))  # b4(同上)
```

运行程序，输出如下：

```
[0.3589779]
[0.746384]
[0.29682708]
[-1.2591735]
Session 2
[0.9770962]
[0.60623896]
[-0.5013621]
```

```
[-1.4085363]
```

上述函数都含有 seed 参数，属于操作级 seed。

在 TensorFlow 中，提供了 range() 函数用于创建数字序列变量，有以下两种形式：

- range(limit, delta=1, dtype=None, name='range')
- range(start, limit, delta=1, dtype=None, name='range')

该数字序列开始于 start 并且将以 delta 为增量扩展到不包括 limit 时的最大值结束，类似 Python 的 range 函数。

【例 1-4】利用 range 函数创建数字序列。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
x=tf.range(8.0, 13.0, 2.0)
y=tf.range(10, 15)
z=tf.range(3, 1, -0.5)
w=tf.range(3)
sess = tf.compat.v1.Session()
sess.run(tf.compat.v1.global_variables_initializer())
print (sess.run(x))
print (sess.run(y))
print (sess.run(z))
print (sess.run(w))
```

运行程序，输出如下：

```
[ 8. 10. 12.]
[10 11 12 13 14]
[3.  2.5 2.  1.5]
[0 1 2]
```

1.5.2　变量的初始化

变量的初始化必须在模型的其他操作运行之前先明确地完成。最简单的方法就是添加一个给所有变量初始化的操作，并在使用模型之前首先运行那个操作。使用 tf.global_variables_initializer() 添加一个操作对变量做初始化。例如：

```
# 创建两个变量
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
...
```

```
# 添加一个操作来初始化变量
init = tf.global_variables_initializer()

# 稍后，当启动模型时
with tf.Session() as sess:
  # Run the init operation.
  sess.run(init)
  ...
  # 使用模型
  ...
```

有时候会需要用另一个变量的初始化值给当前变量初始化。由于 tf.global_variables_initializer() 是并行地初始化所有变量，所以用其他变量的值初始化一个新的变量时，使用其他变量的 initialized_value() 属性，可以直接把已初始化的值作为新变量的初始值，或者把它当作 tensor 计算得到一个值赋予新变量。例如：

```
# 创建一个变量并赋予随机值
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
# 创建另一个变量，使它们的权值相同
w2 = tf.Variable(weights.initialized_value(), name="w2")
# 创建另一个两倍于权值的变量
w_twice = tf.Variable(weights.initialized_value() * 0.2, name="w_twice")
```

assign() 函数也有初始化的功能，tf 中 assign() 函数可用于对变量进行更新，可以更新变量的 value 和 shape。其涉及以下函数：

- tf.assign(ref, value, validate_shape = None, use_locking = None, name=None)
- tf.assign_add(ref, value, use_locking = None, name=None)
- tf.assign_sub(ref, value, use_locking = None, name=None)
- tf.variable.assign(value, use_locking=False)
- tf.variable.assign_add(delta, use_locking=False)
- tf.variable.assign_sub(delta, use_locking=False)

这 6 个函数本质上是一样的，都是用来对变量值进行更新，其中 tf.assign() 可以更新变量的 shape，它是用 value 的值赋给 ref，这种赋值会覆盖原来的值，更新但不会创建一个新的 tensor。tf.assign_add() 相当于用 ref=ref+value 更新 ref。tf.assign_sub() 相当于用 ref=ref-value 更新 ref。tf.variable.assign() 相当于 tf.assign(ref,value)，tf.variable.assign_add() 和 tf.variable.assign_sub() 同理。

tf.assign() 函数的语法格式为：

```
    tf.assign(ref, value, validate_shape = None, use_locking = None,
name=None)
```

其中，参数含义如下。

- ref：一个可变的张量。应该来自变量节点，节点可能未初始化，参考例 1-5。
- value：张量，必须具有与 ref 相同的类型，是要分配给变量的值。
- validate_shape：一个可选的 bool，默认为 True。如果为 True，则操作将验证 value 的形状是否与分配给的张量的形状相匹配；如果为 False，ref 将对“值”的形状进行引用。
- use_locking：一个可选的 bool，默认为 True。如果为 True，则被锁进行保护；否则，该行为是未定义的，但可能会显示较少的争用。
- name：操作的名称（可选）。
- 返回：返回一个在赋值完成后将保留 ref 新值的张量。

下面通过实例演示说明变量的初始化应用。

【例 1-5】变量的初始化应用演示。

说明：assign 操作会初始化相关的节点，并不需要 tf.global_variables_initializer() 初始化，但是并非所有的节点都会被初始化。

```
    import TensorFlow as tf
    import numpy as np
    tf.compat.v1.disable_eager_execution()
    weights=tf.Variable(tf.compat.v1.random_normal([1,2],stddev=0.35),name=
"weights")
    biases=tf.Variable(tf.zeros([3]),name="biases")
    x_data = np.float32(np.random.rand(2, 3)) # 随机输入 2 行 3 列的数据
    y = tf.matmul(weights, x_data) + biases

    update=tf.compat.v1.assign(weights,tf.compat.v1.random_normal([1,2],
stddev=0.50))
    with tf.compat.v1.Session() as sess:
        for _in range(2):
            sess.run(update)
            print(sess.run(weights))# 正确，因为 assign 操作会初始化相关的节点
            print(sess.run(y))# 错误，因为使用了未初始化的 biases 变量
```

1.6　矩阵的操作

理解 TensorFlow 如何计算（操作）矩阵，对于理解计算图中数据的流动来说非常重要。

许多机器学习算法依赖矩阵操作。在 TensorFlow 中，矩阵计算是相当容易的。在下面的所有例子中都会创建一个图会话，代码为：

```
import TensorFlow as tf
sess=tf.Session()
```

1.6.1 矩阵的生成

这部分主要介绍如何生成矩阵，包括全 0 矩阵、全 1 矩阵、随机数矩阵、常数矩阵等。

（1）tf.ones()| tf.zeros() 函数。

这两个函数的用法类似，都是产生尺寸为 shape 的张量（tensor），它们的语法格式为：

```
tf.ones(shape,type=tf.float32,name=None)
tf.zeros([2, 3], int32)
```

【例 1-6】产生大小为 2 × 3 的全 1 矩阵与全 0 矩阵。

```
import TensorFlow.compat.v1 as tf
tf.compat.v1.disable_eager_execution()
sess=tf.compat.v1.Session()
sess = tf.compat.v1.InteractiveSession()
x = tf.ones([2, 3], "int32")
print(sess.run(x))
y = tf.zeros([2, 3], "int32")
print(sess.run(y))
```

运行程序，输出如下：

```
[[1 1 1]
 [1 1 1]]
[[0 0 0]
 [0 0 0]]
```

（2）tf.ones_like() | tf.zeros_like() 函数。

这两个函数用于新建一个与给定 tensor 的类型、大小一致的 tensor，其所有元素为 1 和 0，它们的语法格式为：

```
tf.ones_like(tensor,dype=None,name=None)
tf.zeros_like(tensor,dype=None,name=None)
```

【例 1-7】利用 ones_like() 函数新建一个类型、大小与给定 tensor 一致的全 1 矩阵。

```
    import TensorFlow as tf
    tf.compat.v1.disable_eager_execution()
    sess = tf.compat.v1.InteractiveSession()

    x = tf.ones([2, 3], "float32")
    print("tf.ones():", sess.run(x))
    tensor = [[1, 2, 3], [4, 5, 6]]
    x = tf.ones_like(tensor)
    print(" 与 ones_like() 给定的 tensor 类型、大小一致的 tensor，其所有元素为 1 和 0",
sess.run(x))
    print(" 创建一个形状大小为 shape 的 tensor，其初始值为 value", sess.run(tf.
fill([2, 3], 2)))
```

运行程序，输出如下：

```
    tf.ones(): [[1. 1. 1.]
     [1. 1. 1.]]
    与 ones_like() 给定 tensor 的类型、大小一致，其所有元素为 1 和 0 [[1 1 1]
     [1 1 1]]
    创建一个形状大小为 shape 的 tensor，其初始值为 value [[2 2 2]
     [2 2 2]]
```

运行程序，输出如下：

```
    [[1 1 1]
     [1 1 1]]
    [[0 0 0]
     [0 0 0]]
```

（3）tf.fill() 函数。

该函数用于创建一个形状大小为 shape 的 tensor，其初始值为 value，其语法格式为：

```
    tf.fill(shape,value,name=None)
```

【例 1-8】利用 fill() 函数创建一个形状为 shape 的矩阵。

```
    import TensorFlow as tf
    tf.compat.v1.disable_eager_execution()
    sess=tf.compat.v1.Session()
    print(sess.run(tf.fill([2,4],3)))
```

运行程序，输出如下：

```
    [[3 3 3 3]
```

```
  [3 3 3 3]]
```

（4）tf.constant() 函数。

该函数用于创建一个常量 tensor，按照给出的 value 来赋值，可以用 shape 指定其形状。value 可以是一个数，也可以是一个 list。如果是一个数，那么这个常量中所有的值都用该数赋值。 如果是一个 list，那么 len(value) 一定要小于或等于 shape 展开后的长度，赋值时，先将 value 中的值逐个存入，不够的部分全部存入 value 的最后一个值。

函数的语法格式为：

```
tf.constant(value,dtype=None,shape=None,name='Const')
```

【例 1-9】利用 constant() 函数创建常数矩阵。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()

a = tf.constant(2, shape=[2])
b = tf.constant(2, shape=[2, 2])
c = tf.compat.v1.constant([1, 2, 3], shape=[6])
d = tf.compat.v1.constant([1, 2, 3], shape=[3, 2])

print("constant 的常量: ", sess.run(a))
print("constant 的常量: ", sess.run(b))
print("constant 的常量: ", sess.run(c))
print("constant 的常量: ", sess.run(d))
```

运行程序，输出如下：

```
constant 的常量:  [2 2]
constant 的常量:  [[2 2]
 [2 2]]
constant 的常量:  [1 2 3 3 3 3]
constant 的常量:  [[1 2]
 [3 3]
 [3 3]]
```

（5）tf.random_normal ()| tf.truncated_normal() | tf.random_uniform() 函数。

这几个函数都用于生成随机数 tenso，尺寸是 shape。

- random_normal：随机数，均值为 mean，标准差为 stddev。
- truncated_normal：截断正态分布随机数，均值为 mean，标准差为 stddev，不过只保

留 [mean-2*stddev,mean+2*stddev] 上的随机数。

• random_uniform：均匀分布随机数，范围为 [minval,maxval]。

它们的语法格式分别为：

```
    tf.random_normal(shape,mean=0.0,stddev=1.0,dtype=tf.float32,seed=None,
name=None)
    tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32,
    seed=None, name=None)
    tf.random_uniform(shape,minval=0,maxval=None,dtype=tf.float32,seed=None,
    name=None)
```

【例 1-10】利用 random_normal() 函数生成随机矩阵。

```
    import TensorFlow as tf
    tf.compat.v1.disable_eager_execution()
    sess = tf.compat.v1.InteractiveSession()
    x = tf.compat.v1.random_normal(shape=[1, 5], mean=0.0, stddev=1.0,
dtype=tf.float32, seed=None, name=None)
    print("打印随机数：", sess.run(x))
    x = tf.compat.v1.truncated_normal(shape=[1, 5], mean=0.0, stddev=1.0,
dtype=tf.float32, seed=None, name=None)
    print("截断正态分布随机数：[mean-2*stddev,mean+2*stddev]", sess.run(x))
    x = tf.compat.v1.random_uniform(shape=[1, 5], minval=0, maxval=None,
dtype=tf.float32, seed=None, name=None)
    print("均匀分布随机数：[minval,maxval]", sess.run(x))
```

运行程序，输出如下：

```
    打印随机数： [[-0.56721544  1.890861    0.85449654  0.67190397 -1.3613324 ]]
    截断正态分布随机数：[mean-2*stddev,mean+2*stddev] [[-0.42709324  0.07788924
-0.422174    1.0733088  -0.31796047]]
    均匀分布随机数：[minval,maxval] [[0.3846115  0.14571834 0.7016494  0.38841534
0.6024381 ]]
```

1.6.2　矩阵的变换

TensorFlow 也提供了相关函数用于实现矩阵的变换，下面分别进行介绍。

（1）tf.shape() 函数。

该函数用于返回张量的形状。但需注意，tf.shape() 函数本身也会返回一个张量，在 tf 中，张量需要用 sess.run(Tensor) 来得到具体的值。其语法格式为：

```
    tf.shape(Tensor)
```

【例 1-11】用 shape() 函数返回矩阵的形状。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
labels = [1, 2, 3]
shape = tf.shape(labels)
print(shape)
print("返回张量的形状：", sess.run(shape))
```

运行程序，输出如下：

```
Tensor("Shape:0", shape=(1,), dtype=int32)
返回张量的形状： [3]
```

（2）tf.expand_dims() 函数。

该函数用于为张量 +1 维，其语法格式为：

```
tf.expand_dims(Tensor, dim)
```

【例 1-12】用 expand_dims() 函数为给定矩阵添加一维。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
labels = [1, 2, 3]
x = tf.expand_dims(labels, 0)
print("为张量 +1 维，但是 X 执行的维度为 0，则不更改 ", sess.run(x))
x = tf.expand_dims(labels, 1)
print("为张量 +1 维，X 执行的维度为 1，则增加一维 ", sess.run(x))
x = tf.expand_dims(labels, -1)
print("为张量 +1 维，但是 X 执行的维度为 -1，则不更改 ", sess.run(x))
```

运行程序，输出如下：

```
为张量 +1 维，但是 X 执行的维度为 0，则不更改 [[1 2 3]]
为张量 +1 维，X 执行的维度为 1，则增加一维 [[1]
 [2]
 [3]]
为张量 +1 维，但是 X 执行的维度为 -1，则不更改 [[1]
 [2]
 [3]]
```

（3）tf.concat() 函数。

该函数将张量沿着指定维数拼接起来，其语法格式为：

```
tf.concat(concat_dim, values, name="concat")
```

【例 1-13】利用 concat() 函数将给定的矩阵进行拼接。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
print("tf.concat 将张量沿着指定维数拼接起来", sess.run(tf.concat([t1, t2], 0)))
print("tf.concat 将张量沿着指定维数拼接起来", sess.run(tf.concat([t1, t2], 1)))
```

运行程序，输出如下：

```
tf.concat 将张量沿着指定维数拼接起来 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
tf.concat 将张量沿着指定维数拼接起来 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

（4）tf.sparse_to_dense() 函数。

该函数将稀疏矩阵转为密集矩阵，其语法格式为：

```
def sparse_to_dense(sparse_indices,
                    output_shape,
                    sparse_values,
                    default_value=0,
                    validate_indices=True,
                    name=None):
```

其中，各参数含义如下。

- sparse_indices：元素的坐标，如 [[0,0],[1,2]] 表示 (0,0) 和 (1,2) 处有值。
- output_shape：得到的密集矩阵的 shape。
- sparse_values：sparse_indices 坐标表示的点的值，可以是 0-D 或者 1-D 张量。若是 0-D，则所有稀疏值都一样；若是 1-D，则 len(sparse_values) 应该等于 len(sparse_indices)。
- default_values：默认点的默认值。

（5）tf.random_shuffle() 函数。

该函数将沿着 value 的第一维进行随机重新排列，其语法格式为：

```
tf.random_shuffle(value,seed=None,name=None)
```

【例 1-14】利用 random_shuffle() 函数对给定的矩阵进行重新排列。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()

a = [[1, 2], [3, 4], [5, 6]]
print("沿着 value 的第一维进行随机重新排列：", sess.run(tf.compat.v1.random_
shuffle(a)))
```

运行程序，输出如下：

```
沿着 value 的第一维进行随机重新排列： [[5 6]
 [3 4]
 [1 2]]
```

（6）tf.argmax()| tf.argmin() 函数。

函数找到给定的张量 tensor，并在 tensor 中指定轴 axis 上的最大值的位置。 tf.argmax()，其语法格式为：

```
tf.argmax(input=tensor,dimention=axis)
tf.argmin（）函数同理
```

【例 1-15】利用 argmax() 函数，寻找给定矩阵行与列的最大值。

```
import numpy as np
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
x = np.array([[3, 1, 2],
              [4, 7, 3],
              [5, 0, 1],
              [2, 4, 6]])
a = tf.argmax(x, axis=0)  # 求各列最大值
b = tf.argmax(x, axis=1)  # 求各行最大值

sess = tf.compat.v1.Session()
print("各列最大值 :",sess.run(a))
print("各行最大值 :",sess.run(b))
```

运行程序，输出如下：

```
各列最大值 : [2 1 3]
各行最大值 : [0 1 0 2]
```

（7）tf.equal() 函数。

该函数用于判断两个 tensor 是否每个元素都相等，返回一个格式为 bool 的 tensor，其语法格式为：

```
tf.equal(x, y, name=None):
```

（8）tf.cast() 函数。

该函数将 x 的数据格式转换成 dtype。例如，原来 x 的数据格式是 Bool，那么将其转换成 float 以后，就能将其转换成 0 和 1 的序列，反之也可以。其语法格式为：

```
cast(x, dtype, name=None)
```

【例 1-16】利用 tf.cast() 函数，将给定的 float 数值转换为 Bool 类型。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()

a = tf.Variable([1, 0, 0, 1, 1])
b = tf.cast(a, dtype=tf.bool)
sess.run(tf.compat.v1.global_variables_initializer())
print("float 的数值转换为 Bool 的类型：", sess.run(b))
```

运行程序，输出如下：

```
float 的数值转换为 Bool 的类型：  [ True False Flse    True  True]
```

（9）tf.matmul() 函数。

该函数用来做矩阵乘法。若 a 为 l × m 的矩阵，b 为 m × n 的矩阵，那么通过 tf.matmul(a,b) 就会得到一个 l × n 的矩阵。不过这个函数还提供了很多额外的功能。函数的语法格式为：

```
matmul(a, b,
           transpose_a=False, transpose_b=False,
           a_is_sparse=False, b_is_sparse=False,
           name=None):
```

可以看到，该函数还提供了 transpose 和 is_sparse 的选项。如果对应的 transpose 项为 True，例如 transpose_a=True，那么 a 在参与运算之前会先转置。如果 a_is_sparse=True，那么 a 会被当作稀疏矩阵来参与运算。

【例 1-17】利用 tf .matmul() 函数，对两矩阵进行相乘操作。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])
c = tf.matmul(a, b)
print("矩阵 a 与 b 相乘为：",sess.run(c))
```

运行程序，输出如下：

```
矩阵 a 与 b 相乘为： [[ 58  64]
 [139 154]]
```

（10）tf.reshape() 函数。

该函数将 tensor 按照新的 shape 重新排列。一般来说，shape 有以下三种用法。

- 如果 shape=[−1]，表示要将 tensor 展开成一个 list。
- 如果 shape=[a,b,c,…]，其中每个 a,b,c…均大于零，那么就是常规用法。
- 如果 shape=[a,−1,c,…]，此时 b=−1，a,c…依然都大于零，这时表示 tf 会根据 tensor 的原尺寸，自动计算 b 的值。

函数的语法格式为：

```
reshape(tensor, shape, name=None)
```

【例 1-18】利用 reshape() 函数对矩阵按照新的形状进行重新排列。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
t = [1, 2, 3, 4, 5, 6, 7, 8, 9]
sess.run(tf.compat.v1.global_variables_initializer())
r = tf.reshape(t, [3, 3])
print("重置为 3X3", sess.run(r))
v = tf.reshape(r, [-1])
print("重置回 1X9", sess.run(v))
h = [[[1, 1, 1],
      [2, 2, 2]],
     [[3, 3, 3],
      [4, 4, 4]],
     [[5, 5, 5],
      [6, 6, 6]]]
# -1 被变成了 't'
print("重置 list", sess.run(tf.reshape(h, [-1])))
```

```
# -1 inferred to be 9:
print("重置 2 维", sess.run(tf.reshape(h, [2, -1])))
# -1 当前被推到维 2 :  (-1 is inferred to be 2)
print("重置 2 维", sess.run(tf.reshape(h, [-1, 9])))
# -1 inferred to be 3:
print("重置 3 维", sess.run(tf.reshape(h, [2, -1, 3])))
```

运行程序，输出如下：

```
重置为 3X3 [[1 2 3]
 [4 5 6]
 [7 8 9]]
重置回 1X9 [1 2 3 4 5 6 7 8 9]
重置 list [1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
重置 2 维 [[1 1 1 2 2 2 3 3 3]
 [4 4 4 5 5 5 6 6 6]]
重置 2 维 [[1 1 1 2 2 2 3 3 3]
 [4 4 4 5 5 5 6 6 6]]
重置 3 维 [[[1 1 1]
  [2 2 2]
  [3 3 3]]
 [[4 4 4]
  [5 5 5]
  [6 6 6]]]
```

1.7　图的实现

首先，要弄清楚机器学习框架所谓的“动态框架”和“静态框架”的含义，支持动态计算图的叫动态框架，支持静态计算图的叫静态框架。当然，也有同时支持动态和静态计算图的框架。

在静态框架中使用的是静态声明策略，计算图的声明和执行是分开的。打个比方，现在要造一栋大楼，需要设计图纸和施工队施工，当设计师在设计图纸的时候，施工队什么也不干，等所有图纸设计完成后，施工队才开始施工，当这两个阶段完全分开进行的时候，这种模式是深度学习静态框架模式。在静态框架运行的方式下，先定义计算执行顺序和内存空间分配策略，然后执行过程按照规定的计算执行顺序和当前需求进行计算，数据就在这张实体计算图中计算和传递。常见的静态框架有 TensorFlow、MXNet、Theano 等。

而动态框架中使用的是动态声明策略，其声明和执行是一起进行的。打个比方，动态声明策略就如同设计师和施工队是一起工作的，设计师说“先打地基”，就会马上设计出

打地基的方案并交给施工队去实施，然后设计师又设计出“铺地板”的方案，再交给施工队按照图纸去实施。这样虚拟计算图和实体计算图的构建就是同步进行的，类似于平时写程序的方式。因为可以实时计划，动态框架可以根据实时需求构建对应的计算图，所以在灵活性上，动态框架会更胜一筹。Torch、DyNet、Chainer 等是动态框架。

动态框架灵活性很好，但有代价，所以在现在流行的程序中，静态框架占比更重。静态框架将声明和执行分开有什么好处呢？最大的好处就是在执行前就知道了所有需要进行的操作，所以可以对图中各节点的计算顺序和内存分配进行合理规划，这样就可以较快地执行所需的计算。但是动态框架在每次规划、分配内存、执行的时候，都只能看到局部需求，所以并不能做出全局最优的规划和内存分配。

有了张量和基于张量的各种操作，之后就需要将各种操作整合起来，输出结果。但不幸的是，随着操作种类和数量的增多，有可能引发各种意想不到的问题，例如多个操作之间应该并行还是顺次执行，如何协同各种不同的底层设备，以及如何避免各种类型的冗余操作等。这些问题有可能拉低整个深度学习网络的运行效率或者引入不必要的 bug，为解决这些问题，计算图应运而生。

使用 TensorFlow 编写的程序主要分为两部分，一部分是构建计算图，另一部分是执行计算图。下面构建一个非常简单的计算图。

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
if __name__ == "__main__":
    a = tf.constant([1.0,2.0],name="a")
    b = tf.constant([2.0,3.0],name="b")
    result = a + b
```

在上面的代码中，TensorFlow 会自动将定义的计算转化成计算图上的节点，系统还会自动维护一个默认的计算图。可以通过下面的代码获取当前默认的计算图：

```
    # 通过 a.graph 获取当前节点所属的计算图
    print(a.graph)
    # <TensorFlow.python.framework.ops.Graph object at  0x000001AE4A2A73C8>
    # 判断当前的张量是否属于默认的计算图
    print(a.graph is tf.get_default_graph())
    # True
```

TensorFlow 提供了 tf.Graph() 方法来产生一个新的计算图，在不同的计算图中张量不会共享。例如：

```
g1 = tf.Graph()
# 将计算图 g1 设置为默认计算图
with g1.as_default():
    # 在计算图 g1 中定义变量 c，并将变量 c 初始化为 0
    c = tf.get_variable("c",initializer=tf.zeros_initializer,shape=(1))
# 定义第二个计算图
g2 = tf.Graph()
# 将计算图 g2 设置为默认计算图
with g2.as_default():
    # 在计算图 g2 中定义变量 c，并将变量 c 初始为 1
    c = tf.get_variable("c",initializer=tf.ones_initializer,shape=(1))

# 在计算图 g1 中读取变量 c
with tf.Session(graph=g1) as sess:
    # 初始化变量
    tf.initialize_all_variables().run()
    with tf.variable_scope("",reuse=True):
        # 在计算图 g1 中，定义变量 c 为 0
        print(sess.run(tf.get_variable("c")))
        #[ 0.]
# 在计算图 g2 中读取变量 c
with tf.Session(graph=g2) as sess:
    # 初始化变量
    tf.initialize_all_variables().run()
    with tf.variable_scope("",reuse=True):
        # 在计算图 g2 中定义变量 c 为 1
        print(sess.run(tf.get_variable("c")))
        #[ 1.]
```

分别在计算图 g1 和 g2 中定义张量 c，在 g1 中初始化为 0，在 g2 中初始化为 1，从上面的代码可以看出，在运行不同的计算图时，张量 c 的值是不同的。所以，在 TensorFlow 中可以通过计算图来隔离张量的运算，除此之外，TensorFlow 还为计算图提供了管理张量的机制，可以设置是在 GPU 上还是在 CPU 上进行，设置使用 GPU 可以加速运行，但需要计算机上有 GPU，例如：

```
g = tf.Graph()
    # 指定计算图 g 在 gpu 0（计算机上有多个 GPU，需要指定）上运行
    with g.device("/gpu:0"):
        result = a + b
```

1.8 会话的实现

TensorFlow 中使用会话（session）来执行定义好的运算，会话拥有并管理 TensorFlow 程序运行时的所有资源，当计算完成之后需要关闭会话来帮助系统回收资源。

可以明确调用会话生成函数和关闭函数：

```
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
# 定义两个向量 a,b
a = tf.constant([1.0, 2.0], name='a')
b = tf.constant([2.0, 3.0], name='b')
result = a+b
sess = tf.compat.v1.Session() # 生成一个会话，通过一个会话 session 计算结果
print(sess.run(result))
sess.close() # 关闭会话
```

运行程序，输出如下：

```
[3. 5.]
```

如果程序在执行中异常退出，可能导致会话不能关闭，这时可以使用 Python 上下文管理器机制，将所有的计算放在 with 的内部，在代码块中执行时就可以保持在某种运行状态，而当离开该代码块时就结束当前状态，省去会话关闭代码：

```
with tf.Session() as sess:
    print(sess.run(result))
    #print(result.eval())  # 这行代码也可以直接计算
```

TensorFlow 不会自动生成默认的会话，需要程序员指定。若将会话指定为默认会话，则 TensorFlow 执行时自动启用此会话：

```
sess = tf.Session()
with sess.as_default():
    print(result.eval()) #tf.Tensor.eval 在默认会话中可直接计算张量的值
```

在使用 Python 编写时，可以使用函数直接构建默认会话：

```
sess = tf.InteractiveSession()
print(result.eval())
sess.close()
```

会话可以通过 ConfigProto Protocol Buffer 进行功能配置，如并行的线程数、GPU 分配

策略、运算超过时间等参数设置，比较常用的是以下两个：

```
config = tf.ConfigProto(allow_soft_placement=True, log_device_placement=True)
sess1 = tf.InteractiveSession(config=config)
sess2 = tf.Session(config=config)
```

第一个 allow_soft_placement 参数，当其为 True 时，在以下任意一个条件成立时，GPU 上的运算可以放到 CPU 上计算。

（1）运算不能在 GPU 上运行。

（2）没有空闲 GPU 可使用。

（3）运算输入包含对 CPU 计算结果的引用。

当其为 True 时，可以使代码的可移植性更强。

第二个 log_device_placement 参数，当其为 True 时，日志中将会记录每个节点被安排在了哪个设备上，但这会增加日志量。

如果上述代码在没有 GPU 的机器上运行，会获得以下输出：

```
Device mapping: no known devices.
```

下面通过一个例子来演示张量、计算图及会话的相关操作。

【例 1-19】会话的实现应用实例。

```
# 张量、计算图及会话的相关操作
import TensorFlow as tf
tf.compat.v1.disable_eager_execution()
#tensor 张量：零阶张量是标量 scalar，一阶张量是向量 vector，n 阶张量理解为 n 维数组
# 张量在 TensorFlow 中不是直接采用数组的形式，只是运算结果的引用，即并没有保存数组，
# 保存的是得到这些数字的计算过程

#tf.constan 是一个计算，结果为一个张量，保存在变量 a 中
a=tf.constant([1.0,2.0],name="a")
b=tf.constant([2.0,3.0],name="b")

result=a+b
print(result)
#result.get_shape 获取张量的维度
print(result.get_shape)

# 当计算图构造完成后，张量可以获得计算结果（张量本身没有存储具体的数字）
# 使用 session 来执行定义好的运算（也就是张量存储了运算的过程，使用 session 执行运算
# 获取结果）
```

```
    # 创建会话
    sess=tf.compat.v1.Session()
    res=sess.run(result)
    print(res)
    # 关闭会话释放本地运行使用到的资源
    sess.close()

    # 也可以使用 Python 上下文管理器机制，把所有的计算放在 with 中，上下文管理器退出时自动
    # 释放所有资源，可以避免忘记用 sess.close() 去释放资源
    with tf.compat.v1.Session() as sess:
        print(sess.run(result))

    #as_default 通过默认的会话计算张量的取值，会话不会自动生成默认的会话，需要手动指定，
    # 指定后可以通过 eval 计算张量的取值
    sess =tf.compat.v1.Session()
    with sess.as_default():
        print(result.eval())
    #ConfigProto 配置需要生成的会话
    #allow_soft_placement GPU 设备相关
    #log_device_palcement 日志相关
    config=tf.compat.v1.ConfigProto(allow_soft_placement=True,
                          log_device_placement=True)
    sess1=tf.compat.v1.InteractiveSession(config=config)
    sess2=tf.compat.v1.Session(config=config)
```

运行程序，输出如下：

```
    Tensor("add_20:0", shape=(2,), dtype=float32)
    <bound method Tensor.get_shape of <tf.Tensor 'add_20:0' shape=(2,)
dtype=float32>>
    [3. 5.]
    [3. 5.]
    [3. 5.]
    Device mapping: no known devices.
    Device mapping: no known devices.
```

1.9 读取数据方式

TensorFlow 可以读取许多常用的标准格式，如列表格式（CSV）、图像文件（JPG 和 PNG 格式）和标准 TensorFlow 格式等。

1.9.1　列表格式

为了读列表格式（CSV），TensorFlow 构建了自己的方法。与其他库（如 Pandas）相比，读取一个简单 CSV 文件的过程稍显复杂。

读取 CSV 文件需要两个步骤。首先，必须创建一个文件名队列对象与将使用的文件列表；然后，创建一个 TextLineReader，使用此行读取器解码 CSV 列，并将其保存于张量中。如果想将同质数据混合在一起，可以使用 pack 方法。

【例 1-20】利用 pack 方法读取列表格式信息。

```
    import TensorFlow as tf
    tf.compat.v1.disable_eager_execution()
    # 将文件名列表传入
    filename_queue = tf.compat.v1.train.string_input_producer(["file0.csv",
"file1.csv"],shuffle=True,num_epochs=2)
    # 采用读文本的 reader
    reader = tf.compat.v1.TextLineReader()
    key, value = reader.read(filename_queue)
    # 默认值是 1.0，这里也默认指定了要读入数据的类型是 float
    record_defaults = [[1.0], [1.0]]
    v1, v2 = tf.compat.v1.decode_csv(
        value, record_defaults=record_defaults)
    v_mul = tf.multiply(v1,v2)
    init_op = tf.compat.v1.global_variables_initializer()
    local_init_op = tf.compat.v1.local_variables_initializer()
    # 创建会话
    sess = tf.compat.v1.Session()
    # 初始化变量
    sess.run(init_op)
    sess.run(local_init_op)
    # 输入数据进入队列
    coord = tf.train.Coordinator()
    threads = tf.compat.v1.train.start_queue_runners(sess=sess, coord=coord)
    try:
        while not coord.should_stop():
            value1, value2, mul_result = sess.run([v1,v2,v_mul])
            print("%f\t%f\t%f"%(value1, value2, mul_result))
    except tf.errors.OutOfRangeError:
        print('Done training -- epoch limit reached')
    finally:
        coord.request_stop()
    # 等待线程结束
```

```
    coord.join(threads)
    sess.close()
```

运行程序，输出如下：

```
    2.000000 2.000000      4.000000
    2.000000 3.000000      6.000000
    3.000000 4.000000      12.000000
    1.000000 2.000000      2.000000
    1.000000 3.000000      3.000000
    1.000000 4.000000      4.000000
    1.000000 2.000000      2.000000
    1.000000 3.000000      3.000000
    1.000000 4.000000      4.000000
    2.000000 2.000000      4.000000
    2.000000 3.000000      6.000000
    3.000000 4.000000      12.000000
    Done training -- epoch limit reached
    Process finished with exit code 0
```

1.9.2 读取图像数据

TensorFlow 能够以图像格式导入数据，这对于面向图像的模型非常有用，因为这些模型的输入往往是图像。TensorFlow 支持的图像格式是 JPG 和 PNG，程序内部以 uint8 张量表示，每幅图像通道都是一个二维张量。

【例 1-21】加载一幅原始图像（见图 1-3），并对其进行一些处理，最后保存。

图 1-3 原始图像

```
    import TensorFlow as tf
    tf.compat.v1.disable_eager_execution()
    sess = tf.compat.v1.Session()
    filename_queue = tf.compat.v1.train.string_input_producer(tf.train.
match_filenames_once("xiaoniao.jpg"))
```

```
    reader = tf.compat.v1.WholeFileReader()
    key, value = reader.read(filename_queue)
    image = tf.compat.v1.image.decode_jpeg(value)
    flipImageUpDown = tf.compat.v1.image.encode_jpeg(tf.image.flip_up_
down(image))
    flipImageLeftRight = tf.compat.v1.image.encode_jpeg(tf.image.flip_left_
right(image))
    tf.compat.v1.initialize_all_variables().run(session=sess)
    coord = tf.compat.v1.train.Coordinator()
    threads = tf.compat.v1.train.start_queue_runners(coord=coord, sess=sess)
    example = sess.run(flipImageLeftRight)
    print example
    file = open("flipImageUpDown.jpg", "wb+")
    file.write(flipImageUpDown.eval(session=sess))
    file.close()
    file.open(flipImageLeftRight.eval(session=sess))
    file.close()
```

运行程序，效果如图 1-4 所示。

图 1-4　原始图像与转变后的图像对比（向下翻转与向左翻转）