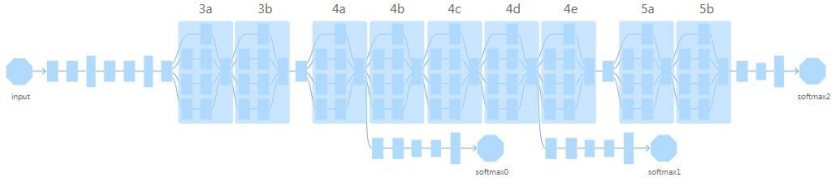
Google TensorFlow™

浙江大学城市学院 ZHEJIANG UNIVERSITY CITY COLLEGE



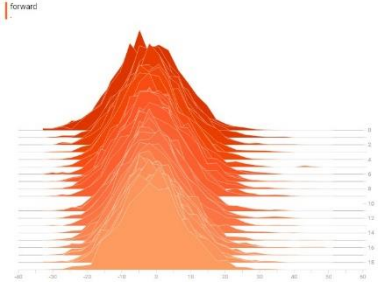
深度学习应用开发
基于TensorFlow的实践

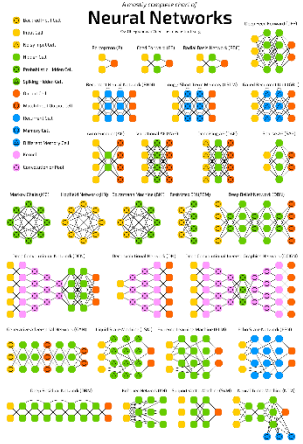
吴明晖 李卓蓉 金苍宏

浙江大学城市学院

计算机与计算科学学院

Dept. of Computer Science
Zhejiang University City College





TensorFlow 2.0 编程基础

浙江大学城市学院
ZHEJIANG UNIVERSITY CITY COLLEGE

A machine learning platform

for everyone

to solve real problems

TensorFlow

- TensorFlow™ 是一个开放源代码软件库，用于进行高性能数值计算
- 借助其灵活的架构，用户可以轻松地将计算工作部署到多种平台（CPU、GPU、TPU）和设备（桌面设备、服务器集群、移动设备、边缘设备等）
- TensorFlow™ 最初是由 Google Brain 团队（隶属于 Google 的 AI 部门）中的研究人员和工程师开发的，可为机器学习和深度学习提供强力支持

TensorFlow编程初体验

浙江大学城市学院 ZHEJIANG UNIVERSITY CITY COLLEGE

In [1]:

```
import tensorflow as tf      # 导入tensorflow
print("Tensorflow版本是：",tf.__version__) #显示当前TensorFlow版本
```

```
Tensorflow版本是： 2.0.0
```

导入Tensorflow库，当前的版本是2.0.0

TensorFlow编程初体验

In [2]:
```
node1 = tf.constant([[3.0,1.5],[2.5,6.0]],tf.float32)  # 创建2个常量
node2 = tf.constant([[4.0,1.0],[5.0,2.5]],tf.float32)
node3 = tf.add(node1, node2)  # 定义加法运算

node3
```

Out[2]:
```
<tf.Tensor: id=2, shape=(2, 2), dtype=float32, numpy=
array([[7. , 2.5],
       [7.5, 8.5]], dtype=float32)>
```

输出的是一个 Tensor

TensorFlow编程初体验


```
[2]: node1

Out[2]: <tf.Tensor: id=0, shape=(2, 2), dtype=float32, numpy=
        array([[3. , 1.5],
               [2.5, 6. ]], dtype=float32)>
```

node1也是一个 Tensor

TensorFlow编程初体验

```
In [3]: print(node3.numpy())   # 输出运算结果Tensor的值
```

```
[[7.  2.5]
 [7.5 8.5]]
```

得到Tensor的值，通过 numpy()方法

TensorFlow 名称的含义

TensorFlow的基本概念

TensorFlow = Tensor + Flow

Tensor 张量

数据结构：多维数组

Flow 流

计算模型：张量之间通过计算而转换的过程

TensorFlow是一个通过**计算图**的形式表述计算的编程系统

每一个计算都是计算图上的一个节点

节点之间的边描述了计算之间的关系

Tensor 张量

张量的概念

```
[2]: node1

Out[2]: <tf.Tensor: id=0, shape=(2, 2), dtype=float32, numpy=
        array([[3. , 1.5],
               [2.5, 6. ]], dtype=float32)>
```

- 在TensorFlow中，所有的数据都通过张量的形式来表示
- 从功能的角度，张量可以简单理解为多维数组

  零阶张量表示**标量**（scalar），也就是**一个数**；

  一阶张量为**向量**（vector），也就是**一维数组**；

  n阶张量可以理解为一个**n维数组**；
- 张量并没有真正保存数字，它**保存的是计算过程**

张量方法和属性

```
<tf.Tensor: id=0, shape=(2, 2), dtype=float32, numpy=
array([[3. , 1.5],
       [2.5, 6. ]], dtype=float32)>
```

标识号（id）

系统自动维护的唯一值

形状（shape）

张量的维度信息

类型（dtype）

每一个张量会有一个唯一的类型,TensorFlow会对参与运算的所有张量进行类型的检查，发现类型不匹配时会报错

值（value）

通过numpy()方法获取，返回Numpy.array类型的数据

TensorFlow编程初体验

In [4]:

```
print(node3.numpy())   # 输出[[7.   2.5]
                       #      [7.5 8.5]]
print(node3.shape)     # 输出(2, 2)，即矩阵的长和宽均为2
print(node3.dtype)     # 输出 <dtype: 'float32'>
```

```
[[7.   2.5]
 [7.5 8.5]]
(2, 2)
<dtype: 'float32'>
```

Tensor的主要方法和属性

张量的形状

三个术语描述张量的维度：**阶**（rank）、**形状**（shape） 、**维数**（dimension number）

阶	形状	维数	例子
0	()	0-D	4
1	(D0)	1-D	[2,3,5]
2	(D0,D1)	2-D	[[2,3],[3,4]]
3	(D0,D1,D2)	3-D	[[[7],[3]],[[2],[4]]]
N	(D0,D1,…,Dn-1)	n-D	形为(D0,D1,…,Dn-1)的张量

表中D0表示第0维元素的个数，Di表示Di维元素的个数

张量的形状

In [5]:

```
scalar = tf.constant(100)
vector = tf.constant([1, 2, 3, 4, 5])
matrix = tf.constant([[1, 2, 3], [4, 5, 6]])
cube_matrix = tf.constant([[[1], [2], [3]], [[4], [5], [6]], [[7], [8], [9]]])

print(scalar.shape)                 # 输出 ()
print(vector.shape)                 # 输出 (5,)
print(matrix.shape)                 # 输出 (2, 3)
print(cube_matrix.get_shape())      # 输出 (3, 3, 1)
```

```
()
(5,)
(2, 3)
(3, 3, 1)
```

查看张量的shape属性还可以通过get_shape()方法来获取。

张量的阶

张量的阶（rank）表征了张量的维度

阶	数学实体	代码示例
0	Scalar	Scalar = 1000
1	Vector	Vector = [2,8,3]
2	Matrix	Matrix = [[4,2,1],[5,3,2],[5,5,6]]
3	3-tensor	Tensor = [[[4],[3],[2]],[[6],[100],[4]],[[5],[1],[4]]]
n	N-tensor	…

获取张量的元素

阶为1的张量等价于向量；

阶为2的张量等价于矩阵，通过 *t[i, j]* 获取元素；

阶为3的张量，通过 *t[i, j, k]* 获取元素；

例：

In [6]:
```
cube_matrix = tf.constant([[[1], [2], [3]], [[4], [5], [6]], [[7], [8], [9]]])
print(cube_matrix.numpy()[1,2,0])  # 输出 6
```

```
6
```

下标从 0 开始

张量的类型

TensorFlow支持不同的类型

实数 tf.float32, tf.float64

整数 tf.int8, tf.int16, tf.int32, tf.int64, tf.uint8

布尔 tf.bool

复数 tf.complex64, tf.complex128

默认类型：

不带小数点的数会被默认为int32

带小数点的会被默认为float32

张量的类型

[7]:
```
a = tf.constant([1, 2])
b = tf.constant([2.0, 3.0])

result = tf.add(a,b)    # 运行报错
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-7-f30deb9a39cb> in <module>
      2 b = tf.constant([2.0, 3.0])
      3
----> 4 result = tf.add(a,b)    # 运行报错
```

运行报错：

InvalidArgumentError: cannot compute Add as input #1(zero-based) was expected to be a int32 tensor but is a float tensor [Op:Add]

TensorFlow会对参与运算的所有张量进行类型的检查，发现类型不匹配时会报错

类型转换

In [8]:
```
a = tf.constant([1, 2])
b = tf.constant([2.0, 3.0])

a = tf.cast(a,tf.float32)  # 数据类型转换

result = tf.add(a,b)
result
```

Out[8]:
```
<tf.Tensor: id=13, shape=(2,), dtype=float32, numpy=array([3., 5.], dtype=float32)>
```

可以通过tf.cast()进行数据类型转换

TensorFlow2的常量与变量

常量 constant

在运行过程中值不会改变的单元

创建语句：

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

在创建常量时只有value值是必填的，dtype等参数可以缺省，会根据具体的value值设置相应的值

创建常量

```
In [9]:  a = tf.constant([1, 2])
         a
```

```
Out[9]: <tf.Tensor: id=14, shape=(2,), dtype=int32, numpy=array([1, 2])>
```

在创建常量时只有value值是必填的，dtype等参数可以缺省，会根据具体的value值设置相应的值

创建常量


```
[10]: b = tf.constant([3, 4],tf.float32)
      b
```

```
Out[10]: <tf.Tensor: id=15, shape=(2,), dtype=float32, numpy=array([3., 4.], dtype=float32)>
```

在创建的同时指定数据类型，在数值兼容的情况下会自动做数据类型转换

创建常量

```
[11]: c = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
      c
```

```
Out[11]: <tf.Tensor: id=18, shape=(2, 3), dtype=int32, numpy=
         array([[1, 2, 3],
                [4, 5, 6]])>
```

如果shape参数值被设定，则会做相应的reshape工作

变量 Variable

在运行过程中值可以被改变的单元

创建语句：

```
tf.Variable (
    initial_value,
    dtype=None,
    shape=None,
    trainable =True
    name='Variable'
)
```

注意V是大写字母

变量 Variable

浙江大学城市学院 ZHEJIANG UNIVERSITY CITY COLLEGE

In [13]:
```
v1 = tf.Variable([1, 2])
v2 = tf.Variable([3, 4],tf.float32)
v1,v2
```

Out[13]:
```
(<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2])>,
 <tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([3, 4])>)
```

变量在创建时必须确定初始值，可以像定义常量一样

变量 Variable

In [14]:
```
c = tf.constant(1)
v = tf.Variable(c)
c,v
```

Out[14]:
```
(<tf.Tensor: id=36, shape=(), dtype=int32, numpy=1>,
 <tf.Variable 'Variable:0' shape=() dtype=int32, numpy=1>)
```

也可以用一个张量作为初始值

变量 Variable

在TensorFlow中变量和普通编程语言中的变量有着**较大区别**

TensorFlow中的变量是一种特殊的设计，是可以被机器优化过程中自动改变值的张量，也可以理解为**待优化的张量**。

在TensorFlow中变量创建后，**一般无需人工进行赋值**，系统会根据算法模型，在训练优化过程中**自动调整变量的值**。

在变量的参数中，trainable参数用来表征当前变量是否需要被自动优化，创建变量对象时默认是启用自动优化标志。

变量的赋值

变量赋值

- 与传统编程语言不同，TensorFlow中的变量定义后，一般**无需**人工赋值，系统会根据算法模型，训练优化过程中**自动调整变量对应的数值**
- 后面在将机器学习模型训练时会更能体会，比如权重Weight变量w，经过多次迭代，会自动调

```
epoch = tf.Variable(0,name='epoch',trainable=False)
```

- 特殊情况需要人工更新的，可用变量赋值语句**assign()**来现实

变量赋值案例

[17]:
```
v = tf.Variable(5)
v.assign(v+1)
v
```

Out[17]: <tf.Variable 'Variable:0' shape=() dtype=int32, numpy=6>

特殊情况需要人工更新的，可用变量赋值语句assign()来现实

变量赋值案例

```
In [18]: v1 = tf.Variable(5)
         v2 = tf.Variable(5)
         v1.assign_add(1)
         v2.assign_sub(1)
         v1,v2

Out[18]: (<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=6>,
          <tf.Variable 'Variable:0' shape=() dtype=int32, numpy=4>)
```

TensorFlow还直接提供了**assign_add()**、**assign_sub()**方法来实现变量的加法和减法值更新

在TensorFlow 2.0中实现TensorFlow 1.x的静态图执行模式

TensorFlow计算模型 - 计算图

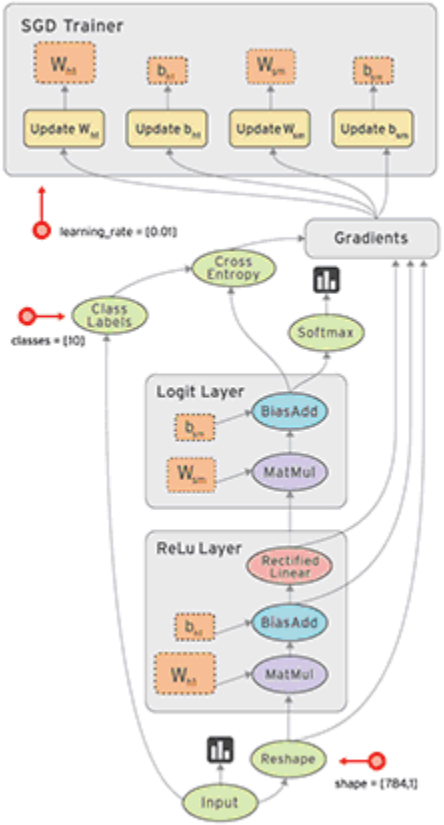
Tensorflow 2 的运行模式

TensorFlow 2代码的执行机制默认采用Eager Execution（动态图执行机制）

TensorFlow 1.x版本代码的执行主要是基于传统的Graph Execution（静态图执行）机制，存在着一定弊端，如入门门槛高、调试困难、灵活性差、无法使用 Python 原生控制语句等

静态图执行模式对于即时执行模式效率会更高，所以通常当模型开发调试完成，部署采用图执行模式会有更高运行效率。在TensorFlow 2里也支持已函数方式调用计算图。

计算图（数据流图）的概念



计算图是一个有向图，由以下内容构成：

- 一组节点，每个**节点**都代表一个**操作**，是一种**运算**
- 一组有向边，每条**边**代表节点之间的**关系（数据传递**和**控制依赖）**

TensorFlow有两种边：

- **常规边（实线）**：代表数据依赖关系。一个节点的运算输出成为另一个节点的输入，两个节点之间有tensor流动（**值传递**）
- **特殊边（虚线）**：不携带值，表示两个节点之间的**控制**相关性。比如，**happens-before关系**，源节点必须在目的节点执行前完成执行

计算图的执行

TensorFlow 1.x版本代码的执行模式缺省是图执行模式。

这种基于静态计算图的图执行模式把程序分为两部分：

1）构建阶段：建立一个“计算图”，通过图的模式来定义数据与操作的执行步骤；

2）执行阶段：建立一个会话，使用会话对象来实现计算图的执行。

计算图的实例

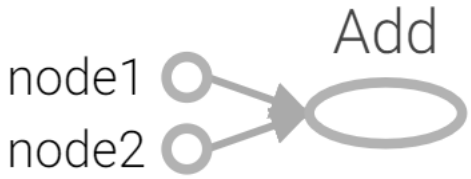
浙江大学城市学院 ZHEJIANG UNIVERSITY CITY COLLEGE

In [2]:
```
# 一个简单计算图
node1 = tf.constant(3.0,tf.float32,name="node1")
node2 = tf.constant(4.0,tf.float32,name="node2")
node3 = tf.add(node1, node2)
```

In [3]:
```
print(node3)
```

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

输出的结果不是一个具体的数字，而是一个张量的结构



node1
Operation: Const

Attributes (2)

dtype	{"type":"DT_FLOAT"}
value	{"tensor": {"dtype":"DT_FLOAT","tensor_shape":{},"float_val":3}}

Inputs (0)

Outputs (0)

Remove from main graph

在TensorFlow 2中实现图执行模式开发

在TensorFlow 2中实现图执行模式开发

- TensorFlow 2 虽然TensorFlow 1.X有较大差异，不能直接兼容TensorFlow 1.X代码。但实际上还是提供了对TensorFlow 1.X 的 API支持
- 原有的TensorFlow 1.X 的 API整理到tensorflow.compat.v1包里去了

TensorFlow 2 中执行或者开发TensorFlow 1.X代码，可以做如下处理：

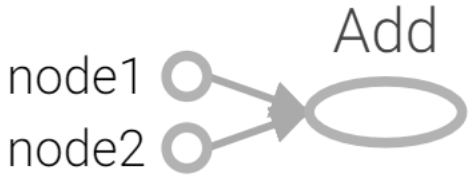
1. 导入TensorFlow时使用 **import tensorflow.compat.v1 as tf** 代替import tensorflow as tf；
2. 执行**tf.disable_eager_execution()** 禁用TensorFlow 2默认的即时执行模式。

体验案例：计算两数之和

In [1]:
```
# 在TensorFlow 2下执行TensorFlow 1.x版本代码
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()      # 改为图执行模式运行
```

In [2]:
```
# 定义了一个简单的“计算图”
node1 = tf.constant(3.0,tf.float32,name="node1")
node2 = tf.constant(4.0,tf.float32,name="node2")
node3 = tf.add(node1, node2)

print(node3)
```

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



由于是图执行模式，这时仅仅是建立了计算图，但它并没有执行

执行计算图

In [4]:

```
sess = tf.Session()     # 建立对话并显示运行结果

print("运行sess.run(node1)的结果：", sess.run(node1))
print("运行sess.run(node2)的结果：", sess.run(node2))
print("运行sess.run(node3)的结果：", sess.run(node3))

sess.close()      # 关闭session
```

```
运行sess.run(node1)的结果： 3.0
运行sess.run(node2)的结果： 4.0
运行sess.run(node3)的结果： 7.0
```

定义好计算图后，需要建立一个会话（Session），使用会话对象来实现计算图的执行。

Session 会话

TensorFlow运行模型 - 会话

会话（session）

会话拥有并管理TensorFlow程序运行时的所有**资源**

当所有计算完成之后需要**关闭会话**帮助系统**回收资源**

会话的模式 1

浙江大学城市学院 ZHEJIANG UNIVERSITY CITY COLLEGE

In [8]:

```
tens1 = tf.constant([1,2,3])     # 定义计算图
sess = tf.Session()     # 创建一个会话

#使用这个创建好的会话来得到关心的运算的结果。比如可以调用 sess.run(tens1)
print(sess.run(tens1))     # 输出结果应该为 [1 2 3]

sess.close()     #关闭会话使得本次运行中使用到的资源可以被释放
```

```
[1 2 3]
```

需要明确调用 Session.close()函数来关闭会话并释放资源

当程序因为异常退出时，关闭会话函数可能就不会被执行从而导致资源泄漏

会话的模式 2

In [9]:

```
tens1 = tf.constant([1,2,3])

sess = tf.Session()
try:
    print(sess.run(tens1))
except:
    print("Exception!")
finally:
    sess.close()     #确保能关闭会话
```

```
[1 2 3]
```

try … except … finally…

会话的模式 3

In [10]:

```
tens1 = tf.constant([1,2,3])

#创建一个会话，并通过Python中的上下文管理器来管理这个会话
with tf.Session() as sess:
    print(sess.run(tens1))

# 不需要再调用 Session.close() 函数来关闭会话
# 当上下文退出时会话关闭和资源释放也自动完成了
```

```
[1 2 3]
```

使用Python的上下文管理器来管理这个会话

指定默认的会话

TensorFlow不会自动生成默认的会话，需要手动指定

当默认的会话被指定之后可以通过 **tf.Tensor.eval** 函数来计算一个张量的取值

In [9]:
```
tens1 = tf.constant([1,2,3])

sess = tf.Session()
with sess.as_default():
    print(tens1.eval())
```

```
[1 2 3]
```

In [15]:
```
tens1 = tf.constant([1,2,3])
sess = tf.Session()

#下面两个命令有相同的功能
print(sess.run(tens1))
print(tens1.eval(session=sess))
```

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

右边代码也可以完成相同的功能

交互式环境下设置默认会话

在交互式环境下，Python脚本或者Jupyter编辑器下，通过设置默认会话来获取张量的取值更加方便

`tf.InteractiveSession` 使用这个函数会自动将生成的会话注册为默认会话

In [11]:
```
tens1 = tf.constant([1,2,3])

sess = tf.InteractiveSession()

print(tens1.eval())
```

```
[1 2 3]
```

变量的初始化

变量的初始化

- 在TensorFlow 2中由于是采用即时执行模式，变量创建后就能直接参与计算，即时得到运算结果。
- 但在TensorFlow 1.X图执行模式中，在模型的其它操作运行之前先**明确地完成变量初始化工作**，否则会报错。
- 对单个变量单独进行初始化可以通过定义并运行**Variable.initializer**操作实现

变量的初始化

In [18]:

```
node1 = tf.Variable(3.0,tf.float32,name="node1")
node2 = tf.Variable(4.0,tf.float32,name="node2")
result = tf.add(node1, node2)

sess = tf.Session()

node1_init = node1.initializer     # 定义单个变量初始化操作
node2_init = node2.initializer

sess.run(node1_init)     # 运行单个变量初始化操作
sess.run(node2_init)

print(sess.run(result))
```

```
7.0
```

对单个变量单独进行初始化可以通过定义并运行**Variable.initializer**操作实现

变量的初始化

In [13]:

```
node1 = tf.Variable(3.0,tf.float32,name="node1")
node2 = tf.Variable(4.0,tf.float32,name="node2")
result = tf.add(node1, node2)

sess = tf.Session()

init = tf.global_variables_initializer()      #定义变量初始化操作
sess.run(init)     #必须先运行这个初始化操作

print(sess.run(result))
```

```
7.0
```

TensorFlow还提供了给所有变量一批进行初始化的语句

tf.global_variables_initializer()

占位符 placeholder

占位符 placeholder

- TensorFlow中的**Variable变量类型**，在定义时需要初始化，但有些变量**定义时并不知道其数值**，只有当真正开始运行程序时，才由外部输入，比如训练数据，这时候需要用到**占位符**
- tf.placeholder**占位符**，是TensorFlow中特有的一种数据结构，类似动态变量，函数的参数、或者C语言或者Python语言中格式化输出时的“%”占位符

占位符 placeholder

- TensorFlow占位符Placeholder，先定义一种数据，其参数为数据的Type和Shape

  占位符Placeholder的函数接口如下：

tf.placeholder(dtype, shape=None, name=None)

```
x = tf.placeholder(tf.float32, [2, 3], name='tx')

# 此代码生成一个2x3的二维数组，矩阵中每个元素的类型都是tf.float32，内部对应的符号名称是tx
```

占位符应用

In [14]:

```
# 在TensorFlow 2下执行TensorFlow 1.x版本代码
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()      # 改为图执行模式运行

a = tf.placeholder(tf.float32, name='a')
b = tf.placeholder(tf.float32, name='b')
c = a+b     # 作用和tf.add(a,b)一样，TensorFlow重载了运算符

a_value = float(input("a = "))  # 从终端读入一个浮点数并放入变量a_
b_value = float(input("b = "))

with tf.Session() as sess:
    # 通过feed_dict的参数传值，按字典格式
    result = sess.run(c, feed_dict={a:a_value, b:b_value})
    print(result)
```

```
a = 5
b = 7
12.0
```

Feed提交数据和Fetch提取数据

Feed提交数据

如果构建了一个包含placeholder操作的计算图，当在session中调用run方法时，placeholder占用的变量必须通过**feed_dict**参数传递进去，否则报错

```
# 在TensorFlow 2下执行TensorFlow 1.x版本代码
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()      # 改为图执行模式运行

a = tf.placeholder(tf.float32, name='a')
b = tf.placeholder(tf.float32, name='b')
c = a+b     # 作用和tf.add(a,b)一样，TensorFlow重载了运算符

a_value = float(input("a = "))  # 从终端读入一个浮点数并放入变量a_
b_value = float(input("b = "))

with tf.Session() as sess:
    # 通过feed_dict的参数传值，按字典格式
    result = sess.run(c, feed_dict={a:a_value, b:b_value})
    print(result)
```

Feed提交数据

多个操作可以通过一次Feed完成执行

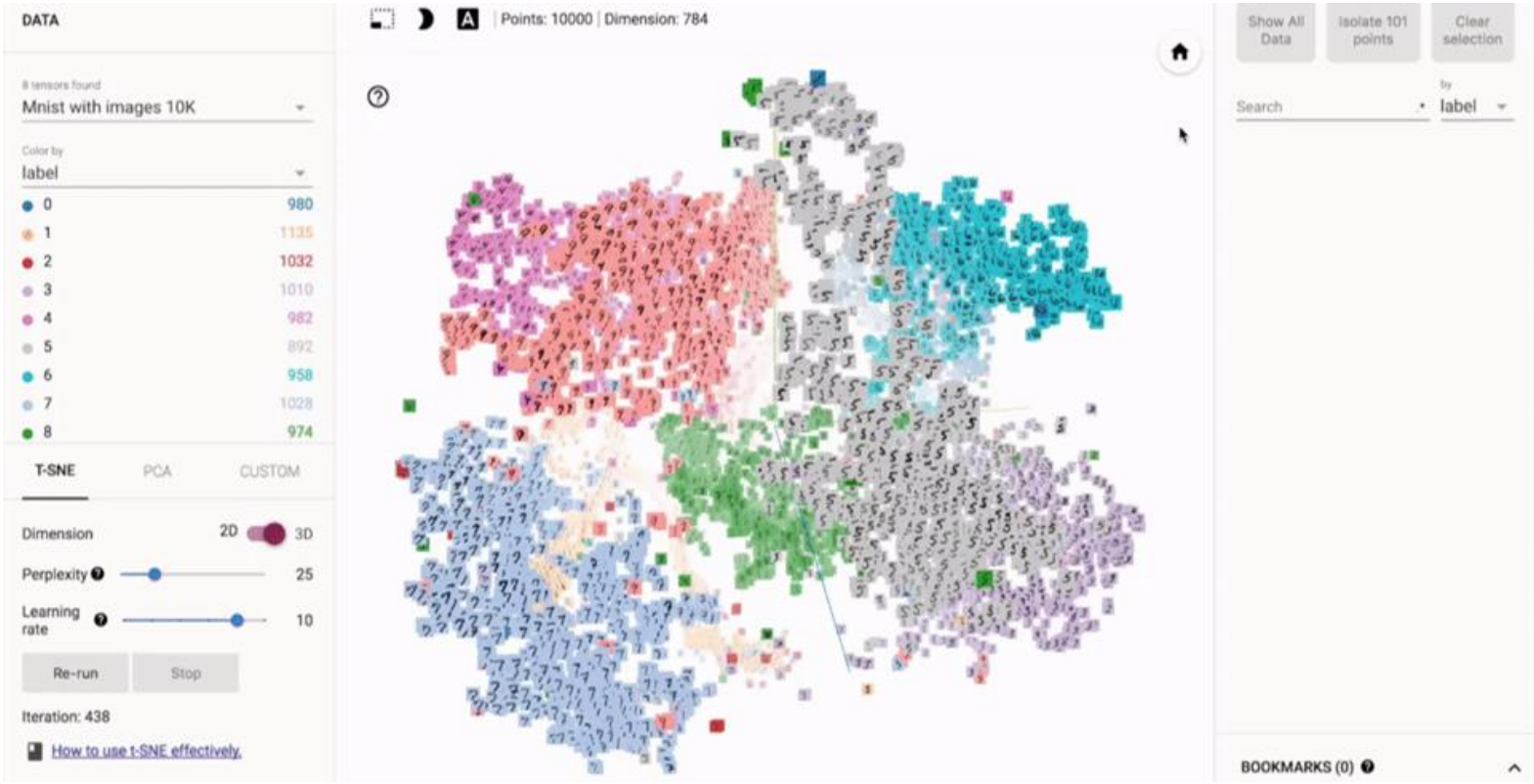
```
# 在TensorFlow 2下执行TensorFlow 1.x版本代码
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()      # 改为图执行模式运行

a = tf.placeholder(tf.float32, name='a')
b = tf.placeholder(tf.float32, name='b')
c = tf.multiply(a, b, name='c')
d = tf.subtract(a, b, name='d')

with tf.Session() as sess:
    #两个操作涉及的占位符一次性Feed，返回的两个值分别赋给两个变量
    rc,rd = sess.run([c,d], feed_dict={a:[8.0,2.0,3.5], b:[1.5,2.0,4.]})
    print("value of c=",rc,"value of d=",rd)
```

```
value of c= [12.   4.  14.] value of d= [ 6.5  0.  -0.5]
```

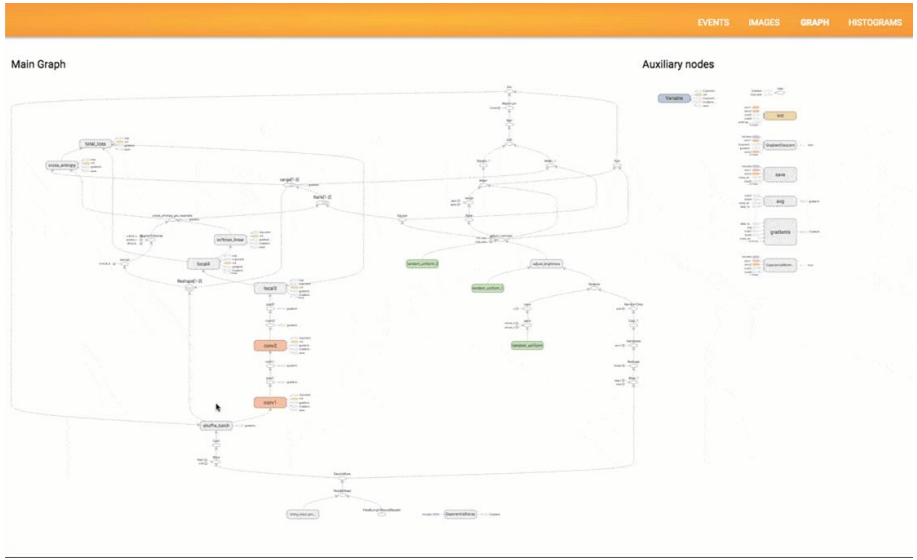
定义占位符时并没有指定形状，所以当实际传递的值是一维数组也是可以的。
运行结果提取出来的两个值组成了一个列表，按顺序分别赋给了两个Python变量。

TensorBoard 可视化初步

TensorBoard：TensorFlow的可视化工具



TensorBoard



计算图可视化

度量可视化

TensorBoard

- TensorBoard是TensorFlow的可视化工具
- 通过TensorFlow程序运行过程中输出的日志文件可视化TensorFlow程序的运行状态
- TensorBoard和TensorFlow程序跑在不同的进程中

案例：在TensorBoard中查看图结构

```
# 在TensorFlow 2下执行TensorFlow 1.x版本代码
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()      # 改为图执行模式运行

tf.reset_default_graph() #清除default graph和不断增加的节点

# 一个简单计算图
node1 = tf.constant(3.0,tf.float32,name="node1")
node2 = tf.constant(4.0,tf.float32,name="node2")
node3 = tf.add(node1, node2)

# logdir改为自己机器上的合适路径
logdir='D:/log'

#生成一个写日志的writer，并将当前的TensorFlow计算图写入日志。
writer = tf.summary.FileWriter(logdir,tf.get_default_graph())
writer.close()
```

案例：在TensorBoard中查看图结构

运行后在指定目录产生了日志文件

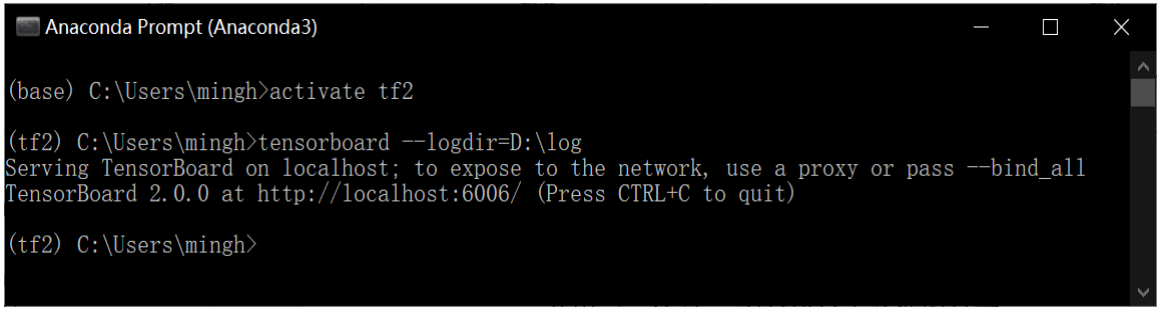
启动TensorBoard

TensorBoard不需要额外安装，在TensorFlow安装时已自动完成

在Anaconda Prompt中

运行TensorBoard，并将日志的地址指向程序日志输出的地址

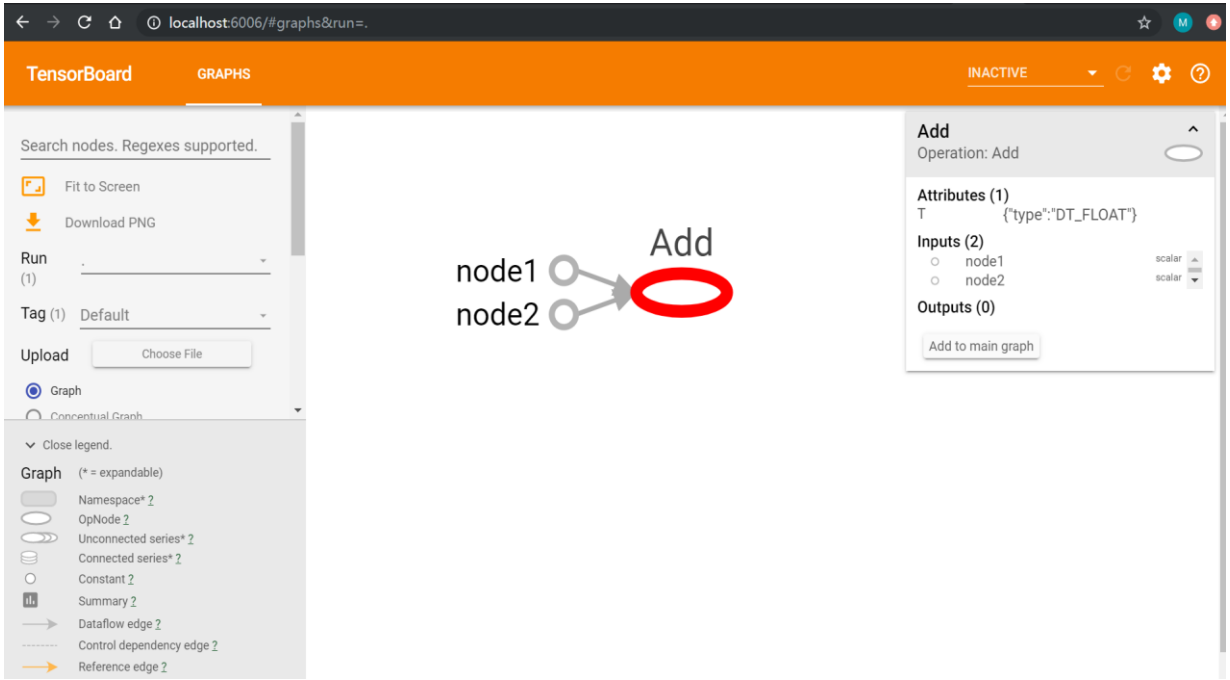
命令： tensorboard --logdir=/path/log



启动服务的端口默认为6006；使用 --port 参数可以改编启动服务的端口

访问TensorBoard

在浏览器访问网址：http://localhost:6006



TensorBoard常用API

API	描述
tf.summary.FileWrite()	创建FileWriter和事件文件，会在logdir中创建一个新的事件文件
tf.summary.FileWriter.add_summary()	将摘要添加到事件文件
tf.summary.FileWriter.add_event()	向事件文件添加一个事件
tf.summary.FileWriter.add_graph()	向事件文件添加一个图
tf.summary.FileWriter.get_logdir()	获取事件文件的路径
tf.summary.FileWriter.flush()	将所有事件都写入磁盘
tf.summary.FileWriter.close()	将事件写入磁盘，并关闭文件操作符
tf.summary.scalar()	输出包含单个标量值的摘要
tf.summary.histogram()	输出包含直方图的摘要
tf.summary.audio()	输出包含音频的摘要
tf.summary.image()	输出包含图片的摘要
tf.summary.merge()	合并摘要，包含所有输入摘要的值