**实验4 TensorFlow基础知识**

**姓名：李坤璘**

**班级学号：2019202216**

1. **实验目的：**

掌握TensorFlow基础知识，为后续深度学习模型构建做准备；

1. **实验条件：**

PC微机一台和深度学习环境。

1. **实验原理：**

TensorFlow 是一个端到端开源机器学习平台。它拥有一个全面而灵活的生态系统，其中包含各种工具、库和社区资源，可助力研究人员推动先进机器学习技术的发展，并使开发者能够轻松地构建和部署由机器学习提供支持的应用。

1. **实验内容：**

【前言：下面分实验截图格式大不相同，是因为内容1、2使用的是pycharm，内容4使用的是jupyter lab，内容3、5、6使用的是vscode里的jupyter插件，是因为做完一二的时候意识到可以直接用jupyter-notebook做实验，然后内容三用了vscode但是总是闪退，又回到anaconda里的jupyter lab，奈何电脑配置太烂内存每次一跑就拉满卡死，无奈又回到vscode……】

**（一）张量**

import tensorflow as tf

import numpy as np

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' # 不输出Info

'''一、创建一些基本张量'''

print('一、创建一些基本张量')

# 1.“标量”（或称“0 秩”张量）。标量包含单个值，但没有“轴”。

rank\_0\_tensor = tf.constant(4)

print(rank\_0\_tensor)



# 2.“向量”（或称“1 秩”张量）就像一个值列表。向量有 1 个轴：

rank\_1\_tensor = tf.constant([2.0, 3.0, 4.0])

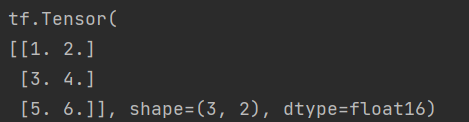
print(rank\_1\_tensor)



# 3.“矩阵”（或称“2 秩”张量）有 2 个轴：

rank\_2\_tensor = tf.constant([[1, 2], [3, 4], [5, 6]], dtype=tf.float16)

print(rank\_2\_tensor)



# 4.张量的轴可能更多，下面是一个包含 3 个轴的张量：

rank\_3\_tensor = tf.constant([

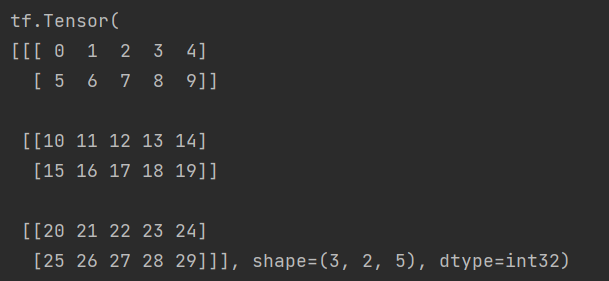
    [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]],

    [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]],

    [[20, 21, 22, 23, 24], [25, 26, 27, 28, 29]]

])

print(rank\_3\_tensor)



'''二、张量的转换'''

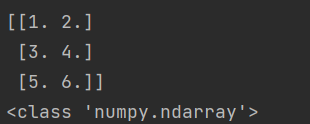
print('二、张量的转换')

# 通过使用 np.array 或 tensor.numpy 方法，将张量转换为 NumPy 数组：

arr1 = np.array(rank\_2\_tensor)

print(arr1)

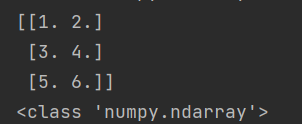
print(type(arr1))



arr2 = rank\_2\_tensor.numpy()

print(arr2)

print(type(arr2))



'''三、张量的运算'''

print('三、张量的运算')

# 1.可以对张量执行基本数学运算

print('1.可以对张量执行基本数学运算:')

a = tf.constant([[1, 2],[3, 4]])

b = tf.ones([2,2],dtype=tf.int32) # 必须类型要相同

print(tf.add(a, b), "\n") # 加法

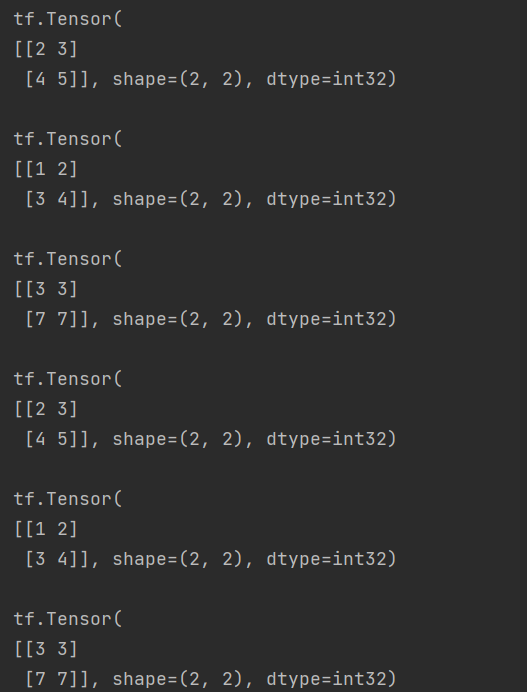
print(tf.multiply(a, b), "\n") # 点乘

print(tf.matmul(a, b), "\n") # 矩阵乘法

print(a + b, "\n") # 加法

print(a \* b, "\n") # 点乘

print(a @ b, "\n") # 矩阵乘法



# 2.各种运算都可以使用张量。

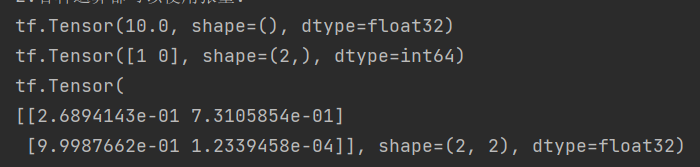
print('2.各种运算都可以使用张量:')

c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

print(tf.reduce\_max(c)) # 最大值

print(tf.math.argmax(c)) # 最大值下角标

print(tf.nn.softmax(c)) # softmax



# 3.转换成张量

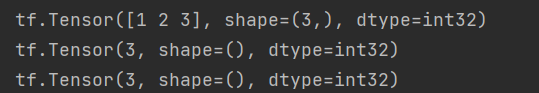
print('3.转换成张量:')

d = np.array([1,2,3])

print(tf.convert\_to\_tensor(d))

print(tf.reduce\_max(d))

print(tf.reduce\_max([1,2,3]))

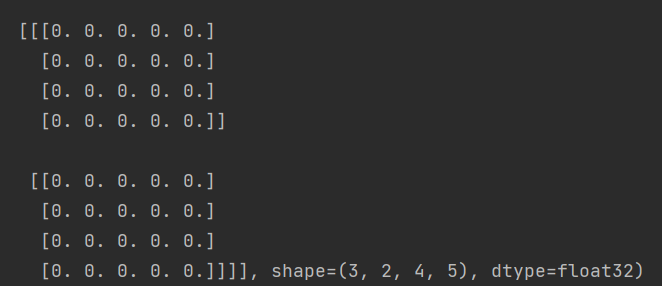


'''四、张量形状简介'''

# 4 秩张量，形状：[3, 2, 4, 5]

rank\_4\_tensor = tf.zeros([3, 2, 4, 5])

print(rank\_4\_tensor)

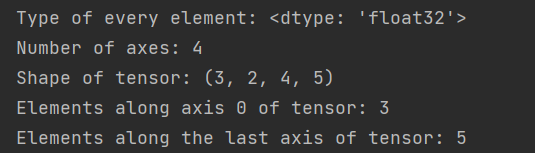


print("Type of every element:", rank\_4\_tensor.dtype) # 数据类型

print("Number of axes:", rank\_4\_tensor.ndim) # 秩4

print("Shape of tensor:", rank\_4\_tensor.shape) # 形状(3, 2, 4, 5)

print("Elements along axis 0 of tensor:", rank\_4\_tensor.shape[0])



# 第一个轴3

print("Elements along the last axis of tensor:", rank\_4\_tensor.shape[-1]) # 最后的轴

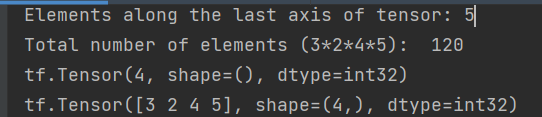
print("Total number of elements (3\*2\*4\*5): ", tf.size(rank\_4\_tensor).numpy()) # 大小

# Tensor.ndim 和 Tensor.shape 特性不返回 Tensor 对象。

# 如果需要 Tensor，请使用 tf.rank 或 tf.shape 函数。

print(tf.rank(rank\_4\_tensor)) # 秩

print(tf.shape(rank\_4\_tensor)) # 形状



# 虽然通常用索引来指代轴，但是您始终要记住每个轴的含义。

# 轴一般按照从全局到局部的顺序进行排序：首先是批次轴，随后是空间维度（长宽），最后是每个位置的特征。

# 这样，在内存中，特征向量就会位于连续的区域。

'''五、张量索引'''

# 1.单轴索引

rank\_1\_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])

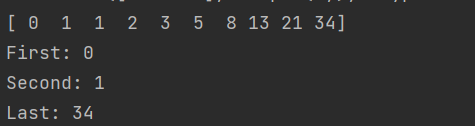
print(rank\_1\_tensor.numpy())

# 使用标量编制索引会移除轴：

print("First:", rank\_1\_tensor[0].numpy())

print("Second:", rank\_1\_tensor[1].numpy())

print("Last:", rank\_1\_tensor[-1].numpy())



# 使用 : 切片编制索引会保留轴：

print("Everything:", rank\_1\_tensor[:].numpy())

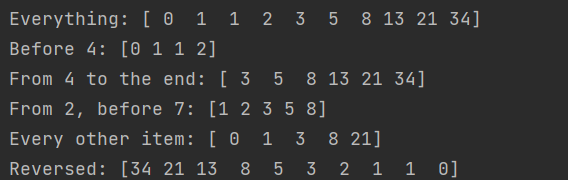
print("Before 4:", rank\_1\_tensor[:4].numpy())

print("From 4 to the end:", rank\_1\_tensor[4:].numpy())

print("From 2, before 7:", rank\_1\_tensor[2:7].numpy())

print("Every other item:", rank\_1\_tensor[::2].numpy())

print("Reversed:", rank\_1\_tensor[::-1].numpy())



# 2.多轴索引

# 对于高秩张量的每个单独的轴，遵循与单轴情形完全相同的规则。

print(rank\_2\_tensor.numpy())

# 为每个索引传递一个整数，结果是一个标量。

print(rank\_2\_tensor[1, 1].numpy())

# 使用整数与切片的任意组合编制索引：

print("Second row:", rank\_2\_tensor[1, :].numpy())

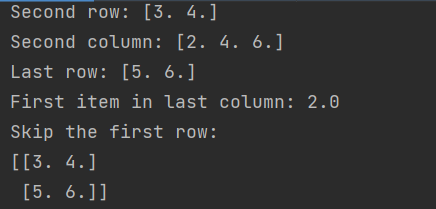
print("Second column:", rank\_2\_tensor[:, 1].numpy())

print("Last row:", rank\_2\_tensor[-1, :].numpy())

print("First item in last column:", rank\_2\_tensor[0, -1].numpy())

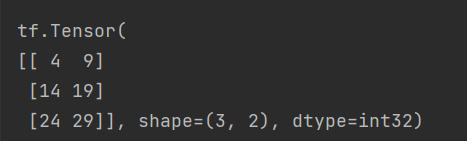
print("Skip the first row:")

print(rank\_2\_tensor[1:, :].numpy(), "\n")



# 3轴张量的示例

print(rank\_3\_tensor[:, :, 4])

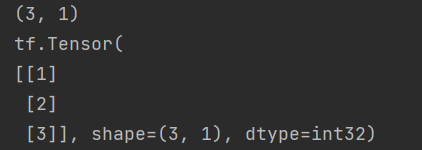


'''# 六、操作形状'''

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

print(x.shape)

print(x)



# 可以直接转换为list

print(x.shape.as\_list())

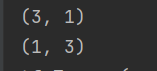


# 通过重构可以改变张量的形状。tf.reshape 运算的速度很快，资源消耗很低，因为不需要复制底层数据。

reshaped = tf.reshape(x, [1, 3])

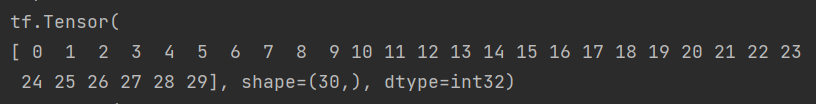
print(x.shape)

print(reshaped.shape)



# 展平张量，则可以看到它在内存中的排列顺序。

print(tf.reshape(rank\_3\_tensor, [-1]))



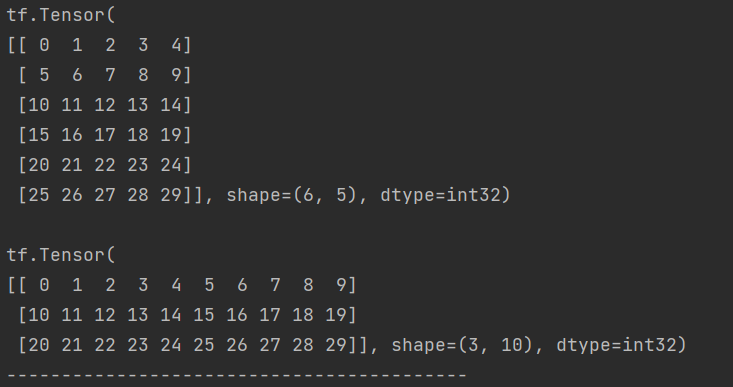
# 一般来说，tf.reshape 唯一合理的用途是用于合并或拆分相邻轴（或添加/移除 1）。

# 对于 3x2x5 张量，重构为 (3x2)x5 或 3x(2x5) 都合理，因为切片不会混淆：

print(tf.reshape(rank\_3\_tensor, [3\*2, 5]), "\n")

print(tf.reshape(rank\_3\_tensor, [3, -1]))

print('------------------------------------------')



# 重构可以处理总元素个数相同的任何新形状，但是如果不遵从轴的顺序，则不会发挥任何作用。

# 利用 tf.reshape 无法实现轴的交换;交换轴，您需要使用 tf.transpose

print(tf.reshape(rank\_3\_tensor, [2, 3, 5]), "\n")

# This is a mess

print(tf.reshape(rank\_3\_tensor, [5, 6]), "\n")

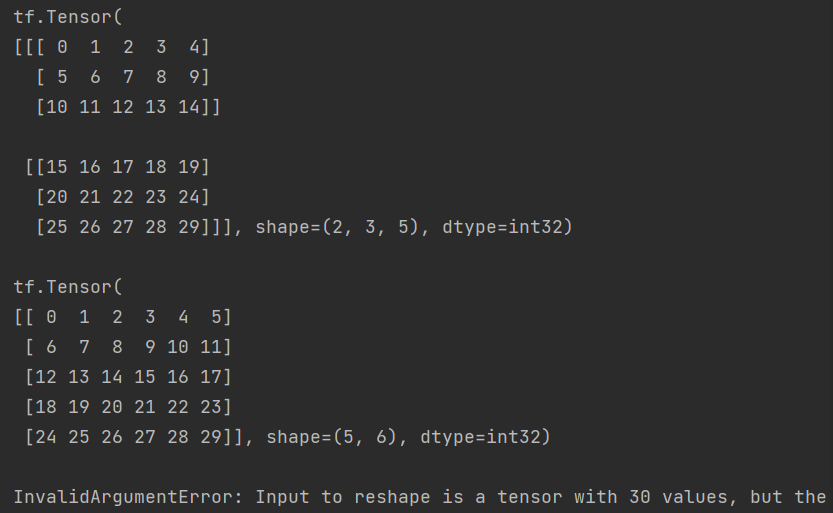
# This doesn't work at all

try:

  tf.reshape(rank\_3\_tensor, [7, -1])

except Exception as e:

  print(f"{type(e).\_\_name\_\_}: {e}")



'''七、DTypes 详解'''

print('七、DTypes 详解')

# 使用 Tensor.dtype 属性可以检查 tf.Tensor 的数据类型。

# 从 Python 对象创建 tf.Tensor 时，您可以选择指定数据类型。

# 如果不指定，TensorFlow 会选择一个可以表示您的数据的数据类型。

# TensorFlow 将 Python 整数转换为 tf.int32，

# 将 Python 浮点数转换为 tf.float32。

# 另外，当转换为数组时，TensorFlow 会采用与 NumPy 相同的规则。

the\_f64\_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)

the\_f16\_tensor = tf.cast(the\_f64\_tensor, dtype=tf.float16)

# 转换类型

the\_u8\_tensor = tf.cast(the\_f16\_tensor, dtype=tf.uint8)

print(the\_u8\_tensor)



'''八、广播'''

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

y = tf.constant(2)

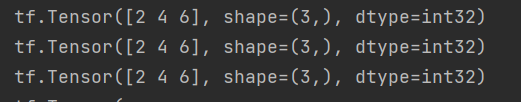
z = tf.constant([2, 2, 2])

# 实现结果相同

print(tf.multiply(x, 2))

print(x \* y)

print(x \* z)



# 实现结果相同

x = tf.reshape(x,[3,1])

y = tf.range(1, 5)

print(x, "\n")

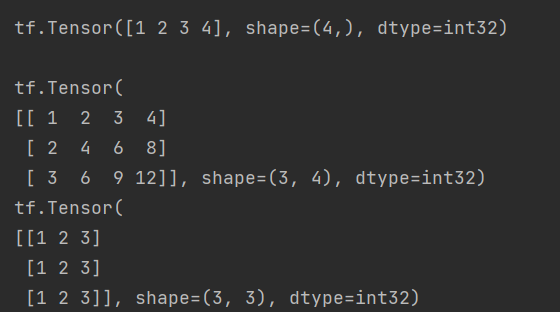
print(y, "\n")

print(tf.multiply(x, y))

# 在大多数情况下，广播的时间和空间效率更高，因为广播运算不会在内存中具体化扩展的张量。

# 使用 tf.broadcast\_to 可以了解广播的运算方式。

print(tf.broadcast\_to(tf.constant([1, 2, 3]), [3, 3]))



'''九、不规则张量'''

# 如果张量的某个轴上的元素个数可变，则称为“不规则”张量。

# 对于不规则数据，请使用 tf.ragged.RaggedTensor。

ragged\_list = [[0, 1, 2, 3],[4, 5],[6, 7, 8],[9]]

try:

  tensor = tf.constant(ragged\_list)

except Exception as e:

  print(f"{type(e).\_\_name\_\_}: {e}")

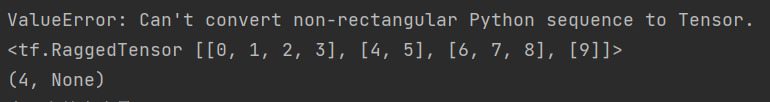
# 应使用 tf.ragged.constant 来创建 tf.RaggedTensor：

ragged\_tensor = tf.ragged.constant(ragged\_list)

print(ragged\_tensor)

# tf.RaggedTensor 的形状将包含一些具有未知长度的轴：

print(ragged\_tensor.shape)



'''十、字符串张量'''

print('十、字符串张量')

# Tensors can be strings, too here is a scalar string.

scalar\_string\_tensor = tf.constant("Gray wolf")

print(scalar\_string\_tensor)

# If you have three string tensors of different lengths, this is OK.

tensor\_of\_strings = tf.constant(["Gray wolf",

                                 "Quick brown fox",

                                 "Lazy dog"])

# Note that the shape is (3,). The string length is not included.

# 在 Python3 中，字符串被视为 Unicode 字符串，即每个字符都是 Unicode 编码。

# 而在 TensorFlow 中，字符串类型的张量是使用字节字符串表示（Byte String），

# 即每个字符是 8 比特的字节表示。因此，当输出字符串类型的张量时，

# 需要以字节字符串的格式输出，这时就会在字符串前面加上 "b" 标识字节字符串。

print(tensor\_of\_strings)

# 如果传递 Unicode 字符，则会使用 utf-8 编码。

print(tf.constant("🥳👍"))

# 在 tf.strings 中可以找到用于操作字符串的一些基本函数，包括 tf.strings.split。

print(tf.strings.split(scalar\_string\_tensor, sep=" "))

# ...but it turns into a `RaggedTensor` if you split up a tensor of strings,

# as each string might be split into a different number of parts.

print(tf.strings.split(tensor\_of\_strings))

# 以及 tf.string.to\_number：

text = tf.constant("1 10 100")

print(tf.strings.to\_number(tf.strings.split(text, " ")))

# 虽然不能使用 tf.cast 将字符串张量转换为数值，但是可以先将其转换为字节，然后转换为数值。

# tf.io 模块包含在数据与字节类型之间进行相互转换的函数，包括解码图像和解析 csv 的函数。

byte\_strings = tf.strings.bytes\_split(tf.constant("Duck"))

byte\_ints = tf.io.decode\_raw(tf.constant("Duck"), tf.uint8)

print("Byte strings:", byte\_strings)

print("Bytes:", byte\_ints)

# 关于Unicode类型字符的分割与解码

unicode\_bytes = tf.constant("アヒル 🦆")

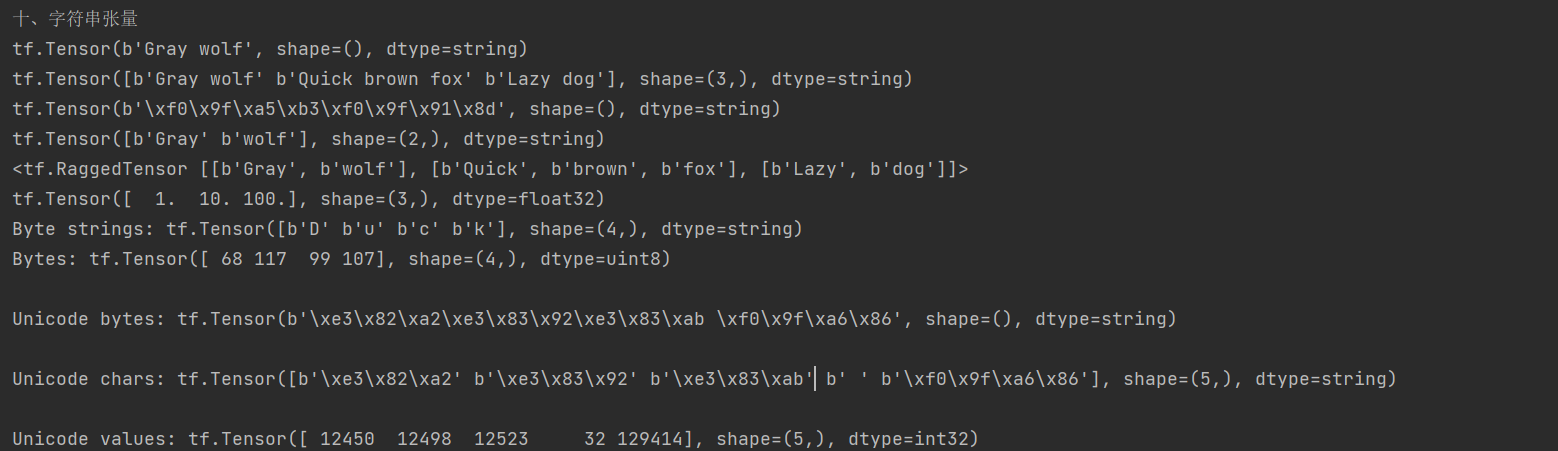
unicode\_char\_bytes = tf.strings.unicode\_split(unicode\_bytes, "UTF-8")

unicode\_values = tf.strings.unicode\_decode(unicode\_bytes, "UTF-8")

print("\nUnicode bytes:", unicode\_bytes)

print("\nUnicode chars:", unicode\_char\_bytes)

print("\nUnicode values:", unicode\_values)



'''十一、稀疏张量'''

# 在某些情况下，数据很稀疏，比如说在一个非常宽的嵌入空间中。

# 为了高效存储稀疏数据，TensorFlow 支持 tf.sparse.SparseTensor 和相关运算。

# Sparse tensors store values by index in a memory-efficient manner

sparse\_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],

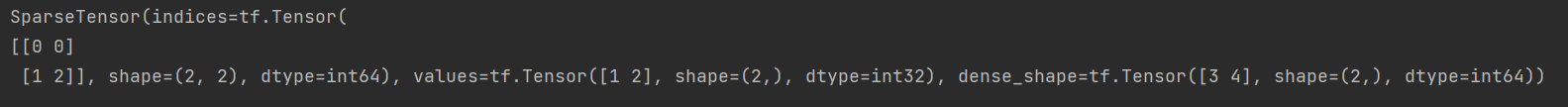
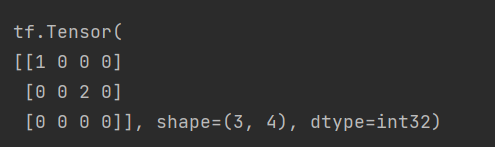
                                       values=[1, 2],

                                       dense\_shape=[3, 4])

print(sparse\_tensor, "\n")

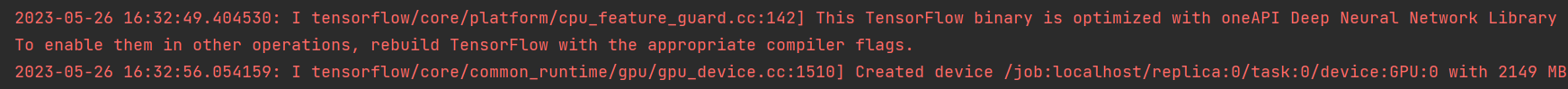
# You can convert sparse tensors to dense

print(tf.sparse.to\_dense(sparse\_tensor))

**（二）变量**

import tensorflow as tf



'''一、创建变量'''

# 要创建变量，请提供一个初始值。tf.Variable 与初始值的 dtype 相同。

my\_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])

my\_variable = tf.Variable(my\_tensor)

# Variables can be all kinds of types, just like tensors

bool\_variable = tf.Variable([False, False, False, True])

complex\_variable = tf.Variable([5 + 4j, 6 + 1j])

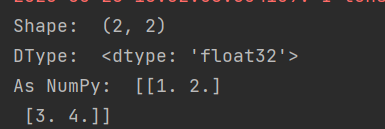
# 变量与张量的定义方式和操作行为都十分相似，实际上，它们都是 tf.Tensor 支持的一种数据结构。

# 与张量类似，变量也有 dtype 和形状，并且可以导出至 NumPy。

print("Shape: ", my\_variable.shape)

print("DType: ", my\_variable.dtype)

print("As NumPy: ", my\_variable.numpy())



'''二、变量运算'''

# 大部分张量运算在变量上也可以按预期运行，不过变量无法重构形状。

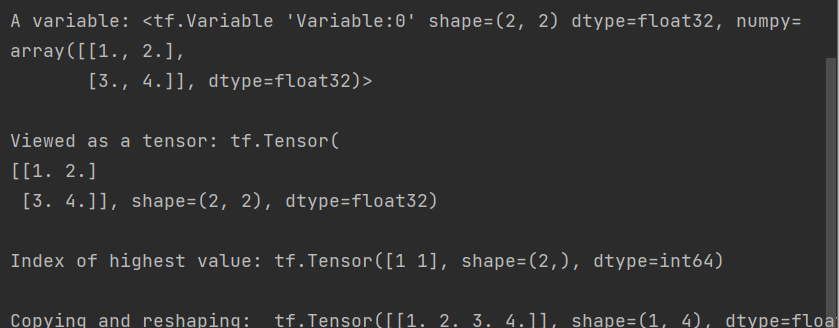
print("A variable:", my\_variable)

print("\nViewed as a tensor:", tf.convert\_to\_tensor(my\_variable))

print("\nIndex of highest value:", tf.math.argmax(my\_variable))

# This creates a new tensor; it does not reshape the variable.

print("\nCopying and reshaping: ", tf.reshape(my\_variable, [1,4]))



'''三、重分配变量'''

# 如上所述，变量由张量提供支持。您可以使用 tf.Variable.assign 重新分配张量。

# 调用 assign（通常）不会分配新张量，而会重用现有张量的内存。

a = tf.Variable([2.0, 3.0])

# This will keep the same dtype, float32

a.assign([1, 2])

# Not allowed as it resizes the variable:

try:

  a.assign([1.0, 2.0, 3.0])

except Exception as e:

  print(f"{type(e).\_\_name\_\_}: {e}")



# 如果在运算中像使用张量一样使用变量，那么通常会对支持张量执行运算。

# 从现有变量创建新变量会复制支持张量。两个变量不能共享同一内存空间。

a = tf.Variable([2.0, 3.0])

# Create b based on the value of a

b = tf.Variable(a)

a.assign([5, 6])

# a and b are different

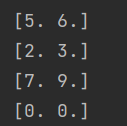
print(a.numpy())

print(b.numpy())

# There are other versions of assign

print(a.assign\_add([2,3]).numpy())  # [7. 9.]

print(a.assign\_sub([7,9]).numpy())  # [0. 0.]



'''四、生命周期、命名和监视'''

# 在基于 Python 的 TensorFlow 中，tf.Variable 实例与其他 Python 对象的生命周期相同。

# 如果没有对变量的引用，则会自动将其解除分配。

a = tf.Variable(my\_tensor, name="Mark")

# A new variable with the same name, but different value

# Note that the scalar add is broadcast

b = tf.Variable(my\_tensor + 1, name="Mark")

# These are elementwise-unequal, despite having the same name

print(a == b)

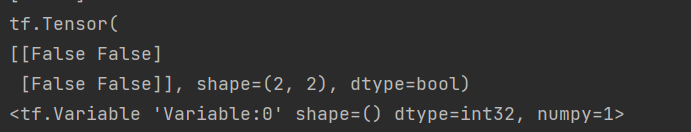
# 虽然变量对微分很重要，但某些变量不需要进行微分。

# 在创建时，通过将 trainable 设置为 False 可以关闭梯度。

# 例如，训练计步器就是一个不需要梯度的变量。

step\_counter = tf.Variable(1, trainable=False)

print(step\_counter)



'''五、放置变量和张量'''

# 如果在有 GPU 和没有 GPU 的不同后端上运行此笔记本，则会看到不同的记录。

with tf.device('CPU:0'):

  # Create some tensors

  a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

  b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

  c = tf.matmul(a, b)

print(c)

# 如果您有多个 GPU 工作进程，但希望变量只有一个副本，则可以这样做

with tf.device('CPU:0'):

  a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

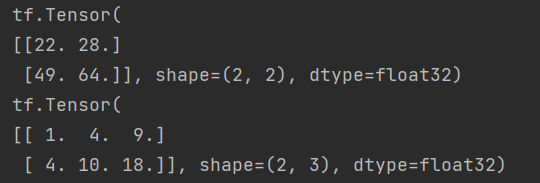
  b = tf.Variable([[1.0, 2.0, 3.0]])

with tf.device('GPU:0'):

  # Element-wise multiply

  k = a \* b

print(k)

  
**（三）自动微分**

**1.设置**

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

**2.梯度带**

TensorFlow 为自动微分提供了 tf.GradientTape API；即计算某个计算相对于某些输入（通常是 tf.Variable）的梯度。TensorFlow 会将在 tf.GradientTape 上下文内执行的相关运算“记录”到“条带”上。TensorFlow 随后会该使用条带通过反向模式微分计算“记录的”计算的梯度。

例如：

x = tf.Variable(3.0)

with tf.GradientTape() as tape:

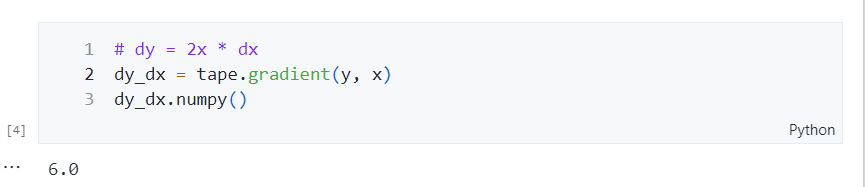
  y = x\*\*2

记录一些运算后，使用 GradientTape.gradient(target, sources) 计算某个目标（通常是损失）相对于某个源（通常是模型变量）的梯度。

# dy = 2x \* dx

dy\_dx = tape.gradient(y, x)

dy\_dx.numpy()



上方示例使用标量，但是 tf.GradientTape 在任何张量上都可以轻松运行：

w = tf.Variable(tf.random.normal((3, 2)), name='w')

b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')

x = [[1., 2., 3.]]

with tf.GradientTape(persistent=True) as tape:

  y = x @ w + b

  loss = tf.reduce\_mean(y\*\*2)

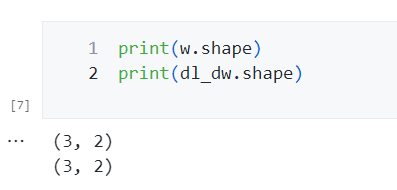
要获得 loss 相对于两个变量的梯度，可以将这两个变量同时作为 gradient 方法的源传递。梯度带在关于源的传递方式上非常灵活，可以接受列表或字典的任何嵌套组合，并以相同的方式返回梯度结构（请参阅 tf.nest）。

[dl\_dw, dl\_db] = tape.gradient(loss, [w, b])

相对于每个源的梯度具有源的形状：

print(w.shape)

print(dl\_dw.shape)



此处也为梯度计算，这一次传递了一个变量字典：

my\_vars = {

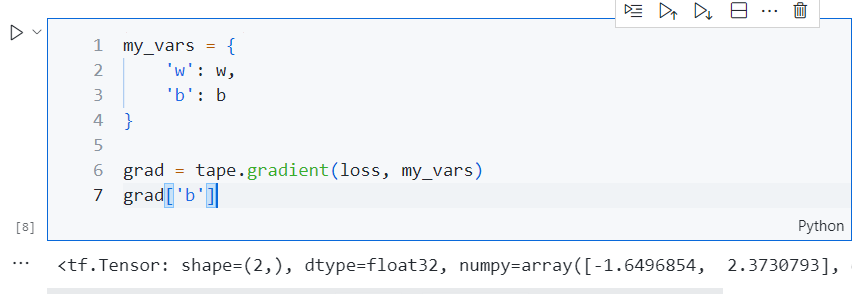
    'w': w,

    'b': b

}

grad = tape.gradient(loss, my\_vars)

grad['b']



**3. 相对于模型的梯度**

通常将 tf.Variables 收集到 tf.Module 或其子类之一（layers.Layer、keras.Model）中，用于设置检查点和导出。

在大多数情况下，需要计算相对于模型的可训练变量的梯度。 由于 tf.Module 的所有子类都在 Module.trainable\_variables 属性中聚合其变量，您可以用几行代码计算这些梯度：

layer = tf.keras.layers.Dense(2, activation='relu')

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

with tf.GradientTape() as tape:

  # Forward pass

  y = layer(x)

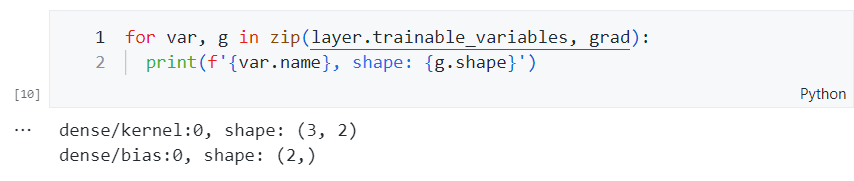
  loss = tf.reduce\_mean(y\*\*2)

# Calculate gradients with respect to every trainable variable

grad = tape.gradient(loss, layer.trainable\_variables)

for var, g in zip(layer.trainable\_variables, grad):

  print(f'{var.name}, shape: {g.shape}')



**4.控制梯度带监视的内容**

默认行为是在访问可训练 tf.Variable 后记录所有运算。原因如下：

· 条带需要知道在前向传递中记录哪些运算，以计算后向传递中的梯度。

· 梯度带包含对中间输出的引用，因此应避免记录不必要的操作。

· 最常见用例涉及计算损失相对于模型的所有可训练变量的梯度。

以下示例无法计算梯度，因为默认情况下 tf.Tensor 未被“监视”，并且 tf.Variable 不可训练：

# A trainable variable

x0 = tf.Variable(3.0, name='x0')

# Not trainable

x1 = tf.Variable(3.0, name='x1', trainable=False)

# Not a Variable: A variable + tensor returns a tensor.

x2 = tf.Variable(2.0, name='x2') + 1.0

# Not a variable

x3 = tf.constant(3.0, name='x3')

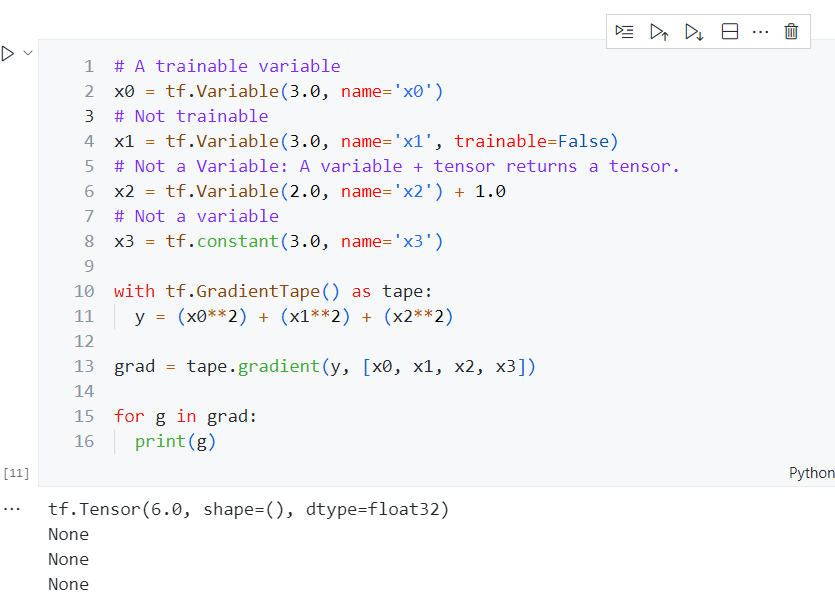
with tf.GradientTape() as tape:

  y = (x0\*\*2) + (x1\*\*2) + (x2\*\*2)

grad = tape.gradient(y, [x0, x1, x2, x3])

for g in grad:

  print(g)



您可以使用 GradientTape.watched\_variables 方法列出梯度带正在监视的变量：

[var.name for var in tape.watched\_variables()]



tf.GradientTape 提供了钩子，让用户可以控制被监视或不被监视的内容。

要记录相对于 tf.Tensor 的梯度，您需要调用 GradientTape.watch(x)：

x = tf.constant(3.0)

with tf.GradientTape() as tape:

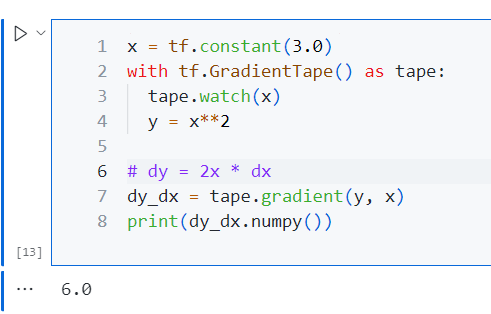
  tape.watch(x)

  y = x\*\*2

# dy = 2x \* dx

dy\_dx = tape.gradient(y, x)

print(dy\_dx.numpy())



相反，要停用监视所有 tf.Variables 的默认行为，请在创建梯度带时设置 watch\_accessed\_variables=False。此计算使用两个变量，但仅连接其中一个变量的梯度：

x0 = tf.Variable(0.0)

x1 = tf.Variable(10.0)

with tf.GradientTape(watch\_accessed\_variables=False) as tape:

  tape.watch(x1)

  y0 = tf.math.sin(x0)

  y1 = tf.nn.softplus(x1)

  y = y0 + y1

  ys = tf.reduce\_sum(y)

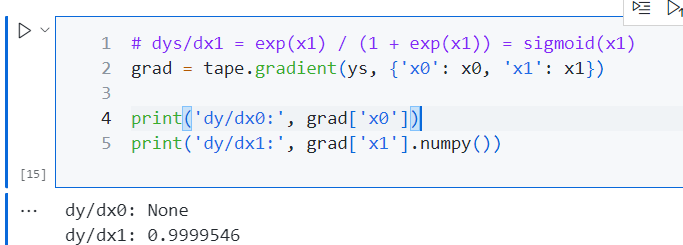
由于 `GradientTape.watch` 未在 `x0` 上调用，未相对于它计算梯度：

# dys/dx1 = exp(x1) / (1 + exp(x1)) = sigmoid(x1)

grad = tape.gradient(ys, {'x0': x0, 'x1': x1})

print('dy/dx0:', grad['x0'])

print('dy/dx1:', grad['x1'].numpy())



**5.中间结果**

您还可以请求输出相对于 `tf.GradientTape` 上下文中计算的中间值的梯度。

x = tf.constant(3.0)

with tf.GradientTape() as tape:

  tape.watch(x)

  y = x \* x

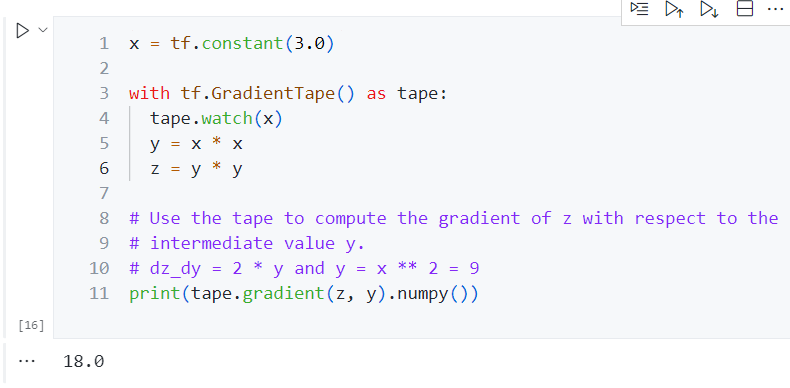
  z = y \* y

# Use the tape to compute the gradient of z with respect to the

# intermediate value y.

# dz\_dy = 2 \* y and y = x \*\* 2 = 9

print(tape.gradient(z, y).numpy())



默认情况下，只要调用 `GradientTape.gradient` 方法，就会释放 `GradientTape` 保存的资源。要在同一计算中计算多个梯度，请创建一个 `persistent=True` 的梯度带。这样一来，当梯度带对象作为垃圾回收时，随着资源的释放，可以对 `gradient` 方法进行多次调用。例如：

x = tf.constant([1, 3.0])

with tf.GradientTape(persistent=True) as tape:

  tape.watch(x)

  y = x \* x

  z = y \* y

print(tape.gradient(z, x).numpy())  # [4.0, 108.0] (4 \* x\*\*3 at x = [1.0, 3.0])

print(tape.gradient(y, x).numpy())  # [2.0, 6.0] (2 \* x at x = [1.0, 3.0])



del tape   # Drop the reference to the tape

**6.非标量目标的梯度**

梯度从根本上说是对标量的运算。

x = tf.Variable(2.0)

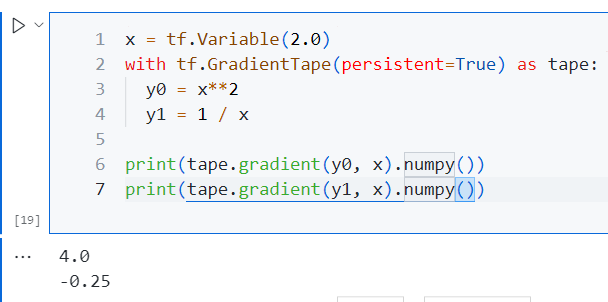
with tf.GradientTape(persistent=True) as tape:

  y0 = x\*\*2

  y1 = 1 / x

print(tape.gradient(y0, x).numpy())

print(tape.gradient(y1, x).numpy())



因此，如果需要多个目标的梯度，则每个源的结果为：

- 目标总和的梯度，或等效

- 每个目标的梯度总和。

x = tf.Variable(2.0)

with tf.GradientTape() as tape:

  y0 = x\*\*2

  y1 = 1 / x

print(tape.gradient({'y0': y0, 'y1': y1}, x).numpy())



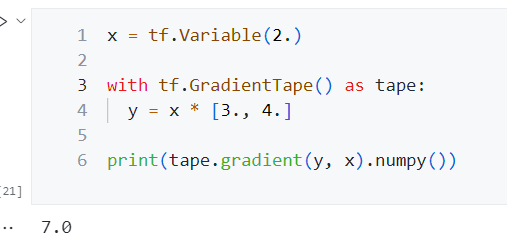
类似地，如果目标不是标量，则计算总和的梯度：

x = tf.Variable(2.)

with tf.GradientTape() as tape:

  y = x \* [3., 4.]

print(tape.gradient(y, x).numpy())



这样一来，就可以轻松获取损失集合总和的梯度，或者逐元素损失计算总和的梯度。

如果每个条目都需要单独的梯度，请参阅雅可比矩阵。

在某些情况下，您可以跳过雅可比矩阵。对于逐元素计算，总和的梯度给出了每个元素相对于其输入元素的导数，因为每个元素都是独立的：

x = tf.linspace(-10.0, 10.0, 200+1)

with tf.GradientTape() as tape:

  tape.watch(x)

  y = tf.nn.sigmoid(x)

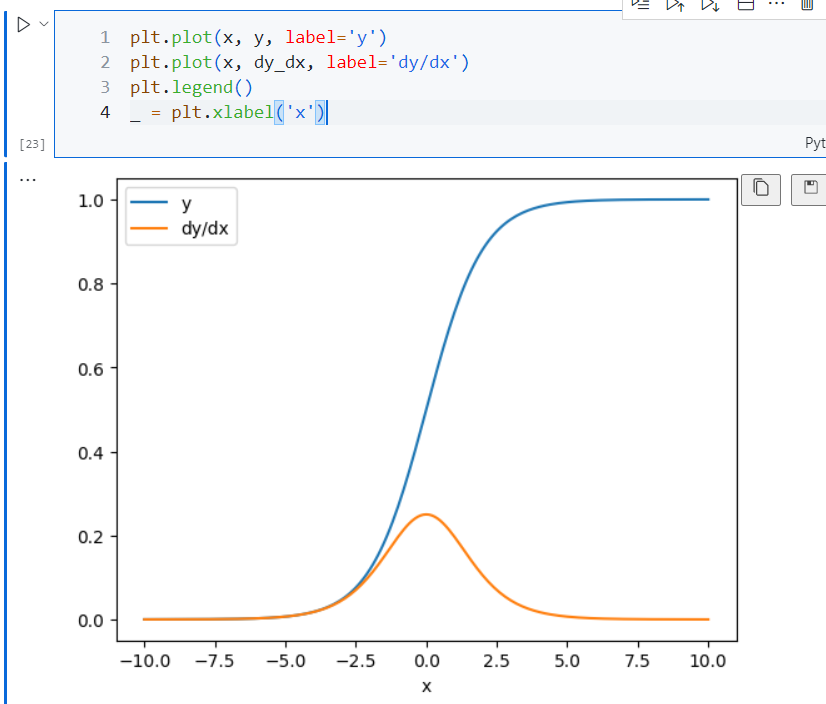
dy\_dx = tape.gradient(y, x)

plt.plot(x, y, label='y')

plt.plot(x, dy\_dx, label='dy/dx')

plt.legend()

\_ = plt.xlabel('x')



**7.控制流**

在执行运算时，由于梯度带会记录这些运算，因此会自然地处理 Python 控制流（例如 if 和 while 语句）。此处，if 的每个分支上使用不同变量。梯度仅连接到使用的变量：

x = tf.constant(1.0)

v0 = tf.Variable(2.0)

v1 = tf.Variable(2.0)

with tf.GradientTape(persistent=True) as tape:

  tape.watch(x)

  if x > 0.0:

    result = v0

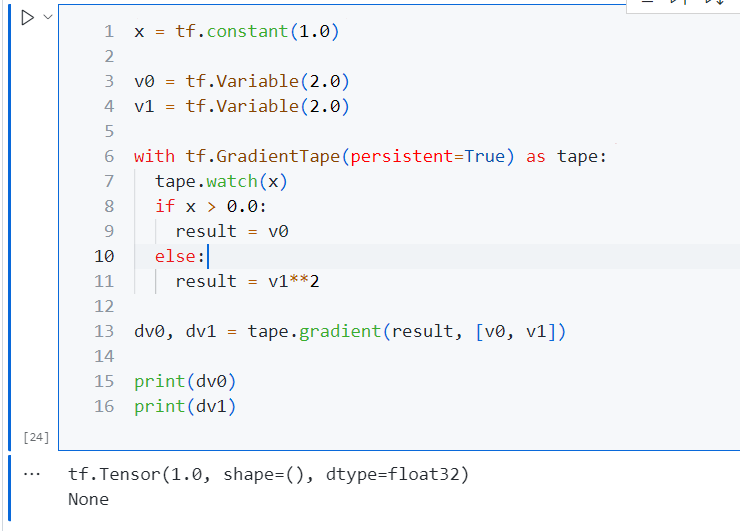
  else:

    result = v1\*\*2

dv0, dv1 = tape.gradient(result, [v0, v1])

print(dv0)

print(dv1)

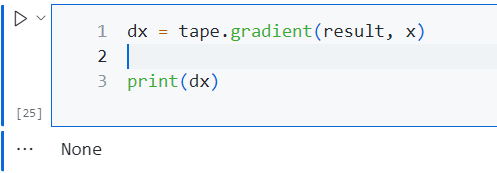


注意，控制语句本身不可微分，因此对基于梯度的优化器不可见。

根据上面示例中 `x` 的值，梯度带将记录 `result = v0` 或 `result = v1\*\*2`。 相对于 `x` 的梯度始终为 `None`。

dx = tape.gradient(result, x)

print(dx)



**8. gradient 返回 None 的情况**

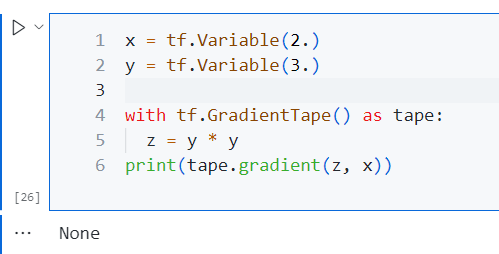
x = tf.Variable(2.)

y = tf.Variable(3.)

with tf.GradientTape() as tape:

  z = y \* y

print(tape.gradient(z, x))



此处 `z` 显然未连接到 `x`，但可以通过几种不太明显的方式将梯度断开。

①使用张量替换变量

在控制梯度带监视内容部分中，梯度带会自动监视 tf.Variable，但不会监视 tf.Tensor。

一个常见错误是无意中将 tf.Variable 替换为 tf.Tensor，而不使用 Variable.assign 更新 tf.Variable。见下例：

x = tf.Variable(2.0)

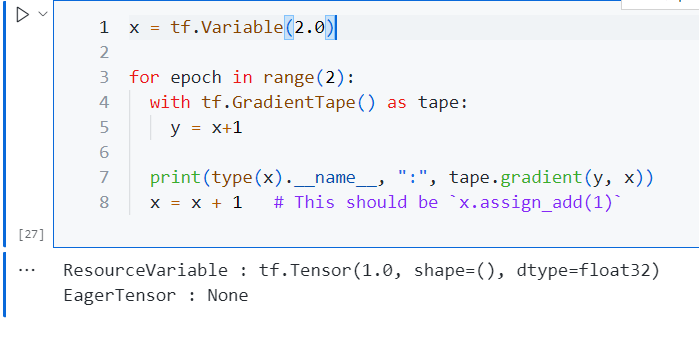
for epoch in range(2):

  with tf.GradientTape() as tape:

    y = x+1

  print(type(x).\_\_name\_\_, ":", tape.gradient(y, x))

  x = x + 1   # This should be `x.assign\_add(1)`



②在 TensorFlow 之外进行了计算

如果计算退出 TensorFlow，梯度带将无法记录梯度路径。例如：

x = tf.Variable([[1.0, 2.0],

                 [3.0, 4.0]], dtype=tf.float32)

with tf.GradientTape() as tape:

  x2 = x\*\*2

  # This step is calculated with NumPy

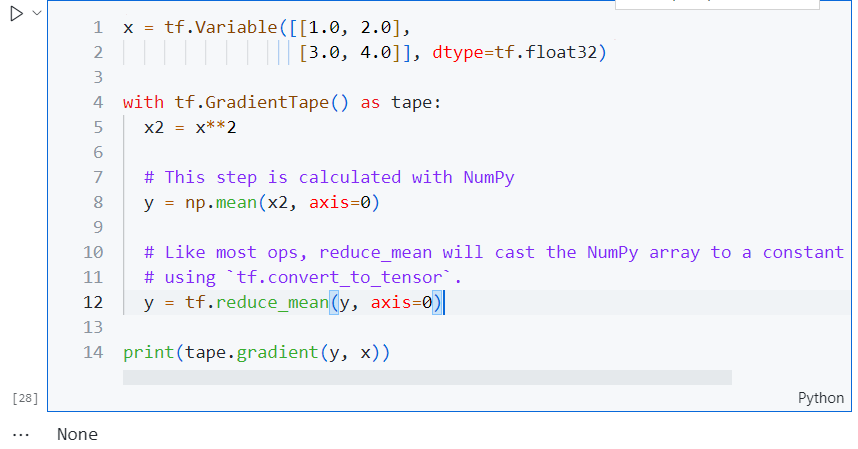
  y = np.mean(x2, axis=0)

  # Like most ops, reduce\_mean will cast the NumPy array to a constant tensor

  # using `tf.convert\_to\_tensor`.

  y = tf.reduce\_mean(y, axis=0)

print(tape.gradient(y, x))



③通过整数或字符串获取梯度

整数和字符串不可微分。如果计算路径使用这些数据类型，则不会出现梯度。

谁也不会期望字符串是可微分的，但是如果不指定 dtype，很容易意外创建一个 int 常量或变量。

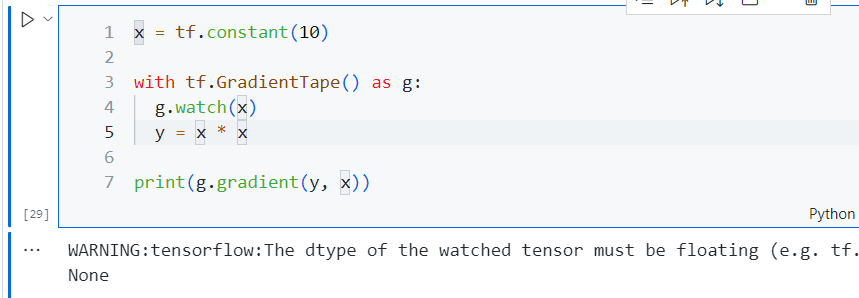
x = tf.constant(10)

with tf.GradientTape() as g:

  g.watch(x)

  y = x \* x

print(g.gradient(y, x))



④通过有状态对象获取梯度

状态会停止梯度。从有状态对象读取时，梯度带只能观察当前状态，而不能观察导致该状态的历史记录。

tf.Tensor 不可变。张量创建后就不能更改。它有一个值，但没有状态。目前讨论的所有运算也都无状态：tf.matmul 的输出只取决于它的输入。

tf.Variable 具有内部状态，即它的值。使用变量时，会读取状态。计算相对于变量的梯度是正常操作，但是变量的状态会阻止梯度计算进一步向后移动。 例如：

x0 = tf.Variable(3.0)

x1 = tf.Variable(0.0)

with tf.GradientTape() as tape:

  # Update x1 = x1 + x0.

  x1.assign\_add(x0)

  # The tape starts recording from x1.

  y = x1\*\*2   # y = (x1 + x0)\*\*2

# This doesn't work.

print(tape.gradient(y, x0))   #dy/dx0 = 2\*(x1 + x0)



**9.未注册梯度**

某些 tf.Operation 被注册为不可微分，将返回 None。还有一些则未注册梯度。

tf.raw\_ops 页面显示了哪些低级运算已经注册梯度。

如果您试图通过一个没有注册梯度的浮点运算获取梯度，梯度带将抛出错误，而不是直接返回 None。这样一来，您可以了解某个环节出现问题。

例如，tf.image.adjust\_contrast 函数封装了 raw\_ops.AdjustContrastv2，此运算可能具有梯度，但未实现该梯度：

image = tf.Variable([[[0.5, 0.0, 0.0]]])

delta = tf.Variable(0.1)

with tf.GradientTape() as tape:

  new\_image = tf.image.adjust\_contrast(image, delta)

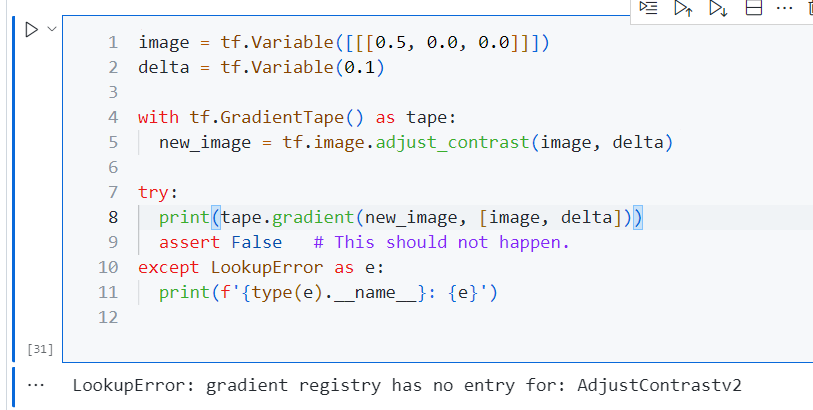
try:

  print(tape.gradient(new\_image, [image, delta]))

  assert False   # This should not happen.

except LookupError as e:

  print(f'{type(e).\_\_name\_\_}: {e}')



**10. 零而不是 None**

在某些情况下，对于未连接的梯度，得到 0 而不是 None 会比较方便。您可以使用 unconnected\_gradients 参数来决定具有未连接的梯度时返回的内容：

x = tf.Variable([2., 2.])

y = tf.Variable(3.)

with tf.GradientTape() as tape:

  z = y\*\*2

print(tape.gradient(z, x, unconnected\_gradients=tf.UnconnectedGradients.ZERO))



**（四）图和函数介绍**

**1.安装**

导入一些所需的库：

import tensorflow as tf

import timeit

from datetime import datetime

**2. 利用计算图**

使用 tf.function 在 TensorFlow 中创建运行计算图，要么作为直接调用，要么作为装饰器。tf.function 将一个常规函数作为输入并返回一个 Function。

Function 是一个 Python 可调用对象，它通过 Python 函数构建 TensorFlow 计算图。

# Define a Python function.

def a\_regular\_function(x, y, b):

  x = tf.matmul(x, y)

  x = x + b

  return x

# `a\_function\_that\_uses\_a\_graph` is a TensorFlow `Function`.

a\_function\_that\_uses\_a\_graph = tf.function(a\_regular\_function)

# Make some tensors.

x1 = tf.constant([[1.0, 2.0]])

y1 = tf.constant([[2.0], [3.0]])

b1 = tf.constant(4.0)

orig\_value = a\_regular\_function(x1, y1, b1).numpy()

# Call a `Function` like a Python function.

tf\_function\_value = a\_function\_that\_uses\_a\_graph(x1, y1, b1).numpy()

assert(orig\_value == tf\_function\_value)

tf.function 适用于一个函数及其调用的所有其他函数：

def inner\_function(x, y, b):

  x = tf.matmul(x, y)

  x = x + b

  return x

# Use the decorator to make `outer\_function` a `Function`.

@tf.function

def outer\_function(x):

  y = tf.constant([[2.0], [3.0]])

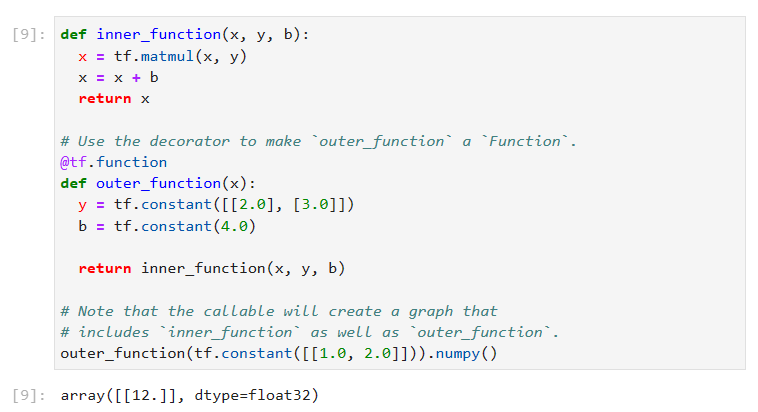
  b = tf.constant(4.0)

  return inner\_function(x, y, b)

# Note that the callable will create a graph that

# includes `inner\_function` as well as `outer\_function`.

outer\_function(tf.constant([[1.0, 2.0]])).numpy()



使用 TensorFlow 编写的任何函数都将包含内置 TF 运算和 Python 逻辑的混合，例如 if-then 子句、循环、break、return、continue 等。虽然 TensorFlow 运算很容易被 tf.Graph 捕获，但特定于 Python 的逻辑需要经过额外的步骤才能成为计算图的一部分。tf.function 使用称为 AutoGraph (tf.autograph) 的库将 Python 代码转换为计算图生成代码。

def simple\_relu(x):

  if tf.greater(x, 0):

    return x

  else:

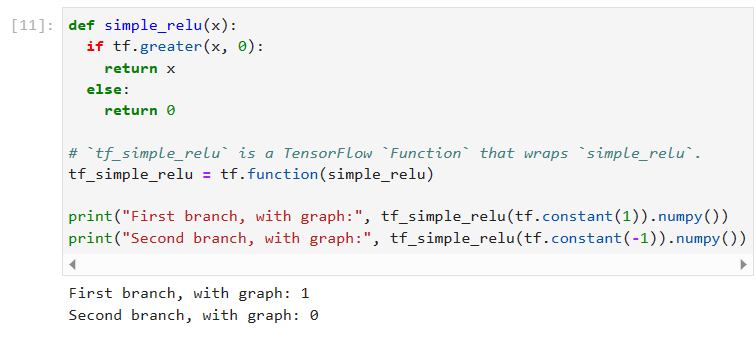
    return 0

# `tf\_simple\_relu` is a TensorFlow `Function` that wraps `simple\_relu`.

tf\_simple\_relu = tf.function(simple\_relu)

print("First branch, with graph:", tf\_simple\_relu(tf.constant(1)).numpy())

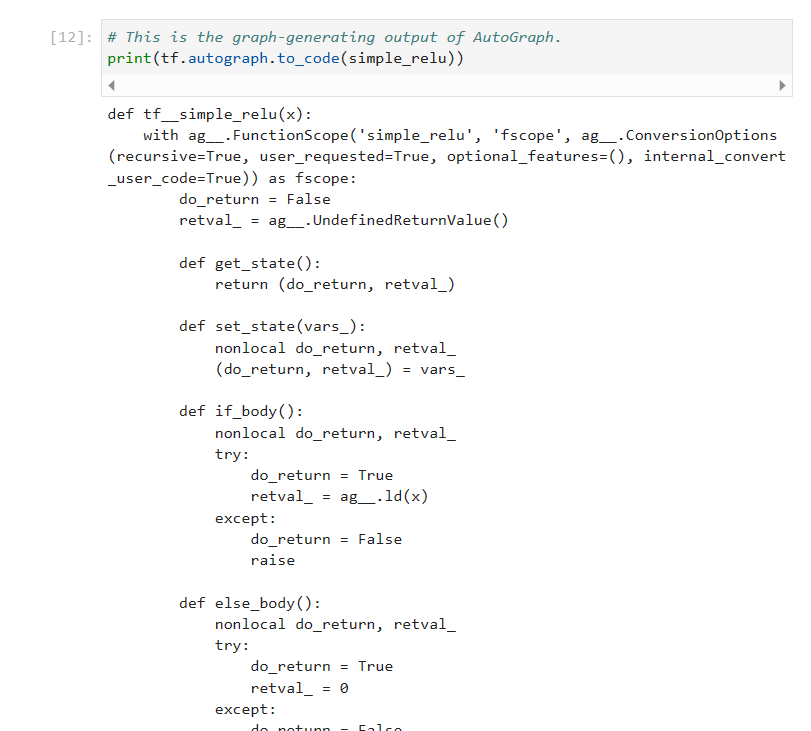
print("Second branch, with graph:", tf\_simple\_relu(tf.constant(-1)).numpy())



虽然不太可能需要直接查看计算图，但可以检查输出以验证确切的结果。

# This is the graph-generating output of AutoGraph.

print(tf.autograph.to\_code(simple\_relu))



# This is the graph itself.

print(tf\_simple\_relu.get\_concrete\_function(tf.constant(1)).graph.as\_graph\_def())



**3.多态性：一个 Function，多个计算图**

tf.Graph 专门用于特定类型的输入（例如，具有特定 dtype 的张量或具有相同 id() 的对象）。

每次使用一组无法由现有的任何计算图处理的参数（例如具有新 dtypes 或不兼容形状的参数）调用 Function 时，Function 都会创建一个专门用于这些新参数的新 tf.Graph。tf.Graph 输入的类型规范被称为它的输入签名或签名。如需详细了解何时生成新的 tf.Graph 以及如何控制它，请转到使用 tf.function 提高性能指南的回溯规则部分。

Function 在 ConcreteFunction 中存储与该签名对应的 tf.Graph。ConcreteFunction 是围绕 tf.Graph 的封装容器。

@tf.function

def my\_relu(x):

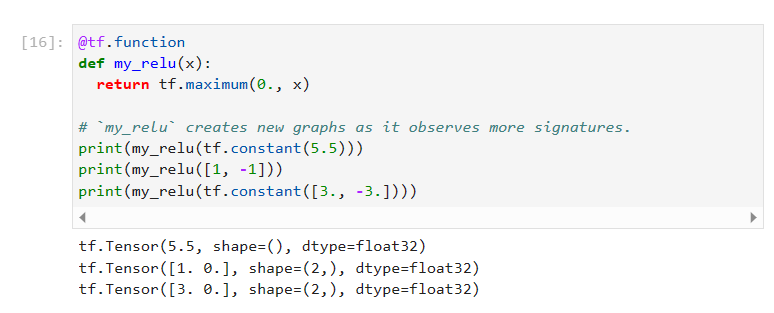
  return tf.maximum(0., x)

# `my\_relu` creates new graphs as it observes more signatures.

print(my\_relu(tf.constant(5.5)))

print(my\_relu([1, -1]))

print(my\_relu(tf.constant([3., -3.])))

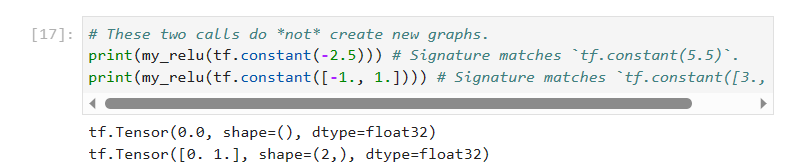


如果已经使用该签名调用了 `Function`，则该 `Function` 不会创建新的 `tf.Graph`。

# These two calls do \*not\* create new graphs.

print(my\_relu(tf.constant(-2.5))) # Signature matches `tf.constant(5.5)`.

print(my\_relu(tf.constant([-1., 1.]))) # Signature matches `tf.constant([3., -3.])`.

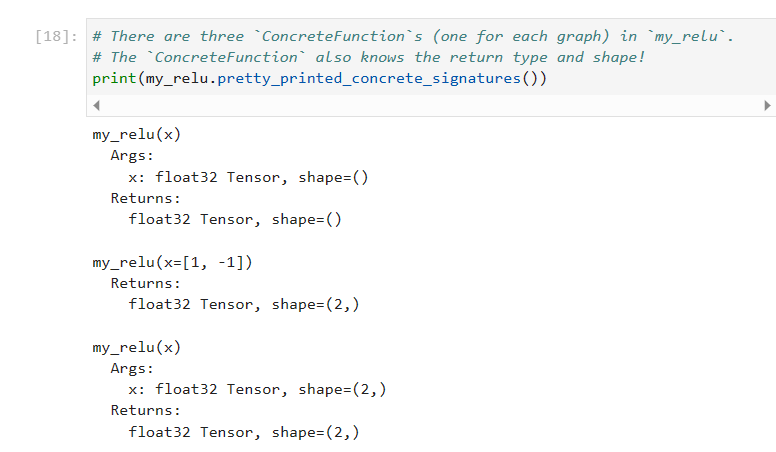


由于它由多个计算图提供支持，因此 `Function` 是多态的。这样，它便能够支持比单个 `tf.Graph` 可以表示的更多的输入类型，并优化每个 `tf.Graph` 来获得更出色的性能。

# There are three `ConcreteFunction`s (one for each graph) in `my\_relu`.

# The `ConcreteFunction` also knows the return type and shape!

print(my\_relu.pretty\_printed\_concrete\_signatures())



**4.使用 tf.function-计算图执行与 Eager Execution**

Function 函数中的代码既能以 Eager 模式执行，也可以作为计算图执行。默认情况下，Function 将其代码作为计算图执行：

@tf.function

def get\_MSE(y\_true, y\_pred):

  sq\_diff = tf.pow(y\_true - y\_pred, 2)

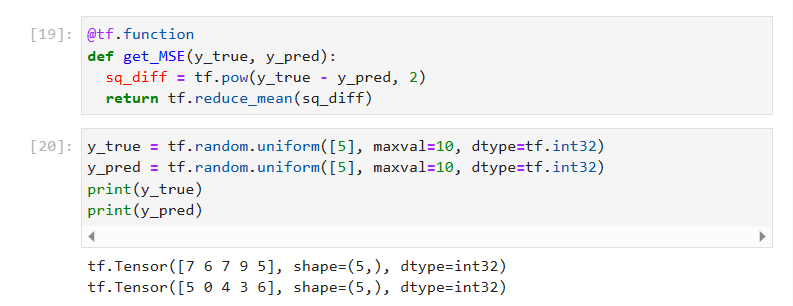
  return tf.reduce\_mean(sq\_diff)

y\_true = tf.random.uniform([5], maxval=10, dtype=tf.int32)

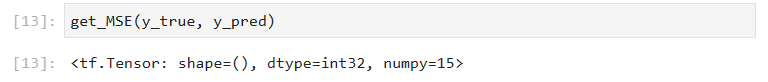
y\_pred = tf.random.uniform([5], maxval=10, dtype=tf.int32)

print(y\_true)

print(y\_pred)



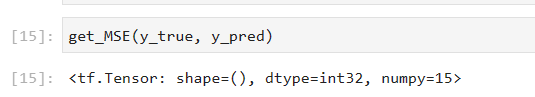
get\_MSE(y\_true, y\_pred)



要验证 Function 计算图是否与其等效 Python 函数执行相同的计算，您可以使用 tf.config.run\_functions\_eagerly(True) 使其以 Eager 模式执行。这是一个开关，用于关闭 Function 创建和运行计算图的能力，无需正常执行代码。

tf.config.run\_functions\_eagerly(True)

get\_MSE(y\_true, y\_pred)



# Don't forget to set it back when you are done.

tf.config.run\_functions\_eagerly(False)

但是，Function 在计算图执行和 Eager Execution 下的行为可能有所不同。Python print 函数是这两种模式不同之处的一个示例。我们看看当您将 print 语句插入到您的函数并重复调用它时会发生什么。

@tf.function

def get\_MSE(y\_true, y\_pred):

  print("Calculating MSE!")

  sq\_diff = tf.pow(y\_true - y\_pred, 2)

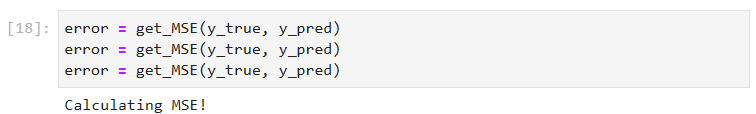
  return tf.reduce\_mean(sq\_diff)

观察打印的内容：

error = get\_MSE(y\_true, y\_pred)

error = get\_MSE(y\_true, y\_pred)

error = get\_MSE(y\_true, y\_pred)



输出很令人惊讶？尽管 get\_MSE 被调用了 3 次，但它只打印了一次。

解释一下，print 语句在 Function 运行原始代码时执行，以便在称为“跟踪”（请参阅 tf.function 指南的跟踪部分）的过程中创建计算图。跟踪将 TensorFlow 运算捕获到计算图中，而计算图中未捕获 print。随后对全部三个调用执行该计算图，而没有再次运行 Python 代码。

作为健全性检查，我们关闭计算图执行来比较：

# Now, globally set everything to run eagerly to force eager execution.

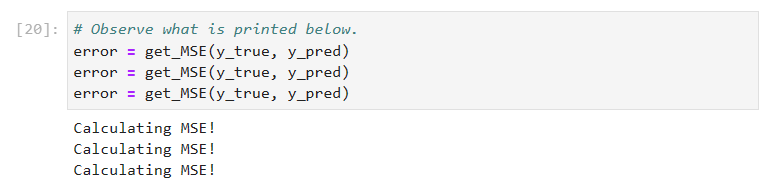
tf.config.run\_functions\_eagerly(True)

# Observe what is printed below.

error = get\_MSE(y\_true, y\_pred)

error = get\_MSE(y\_true, y\_pred)

error = get\_MSE(y\_true, y\_pred)



tf.config.run\_functions\_eagerly(False)

print 是 Python 的副作用，在将函数转换为 Function 时，您还应注意其他差异。请在使用 tf.function 提升性能指南中的限制部分中了解详情。

注：如果您想同时在 Eager Execution 和计算图执行中打印值，请改用 tf.print。

**5. 非严格执行**

计算图执行仅执行产生可观察效果所需的运算，这包括：

· 函数的返回值

· 已记录的著名副作用，例如：

· 输入/输出运算，如 tf.print

· 调试运算，如 tf.debugging 中的断言函数

· tf.Variable 的突变

这种行为通常称为“非严格执行”，与 Eager Execution 不同，后者会分步执行所有程序运算，无论是否需要。

特别是，运行时错误检查不计为可观察效果。如果一个运算因为不必要而被跳过，它不会引发任何运行时错误。

在下面的示例中，计算图执行期间跳过了tf.gather不会像在 Eager Execution 中那样引发运行时错误 InvalidArgumentError。切勿依赖执行计算图时引发的错误。

def unused\_return\_eager(x):

  # Get index 1 will fail when `len(x) == 1`

  tf.gather(x, [1]) # unused

  return x

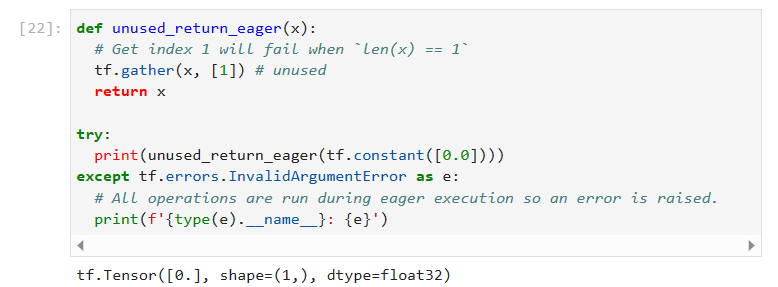
try:

  print(unused\_return\_eager(tf.constant([0.0])))

except tf.errors.InvalidArgumentError as e:

  # All operations are run during eager execution so an error is raised.

  print(f'{type(e).\_\_name\_\_}: {e}')



@tf.function

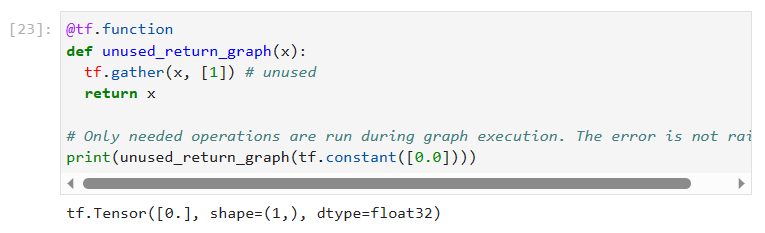
def unused\_return\_graph(x):

  tf.gather(x, [1]) # unused

  return x

# Only needed operations are run during graph execution. The error is not raised.

print(unused\_return\_graph(tf.constant([0.0])))



**6.见证加速**

tf.function 通常可以提高代码的性能，但加速的程度取决于您运行的计算种类。小型计算可能以调用计算图的开销为主。您可以按如下方式衡量性能上的差异：

x = tf.random.uniform(shape=[10, 10], minval=-1, maxval=2, dtype=tf.dtypes.int32)

def power(x, y):

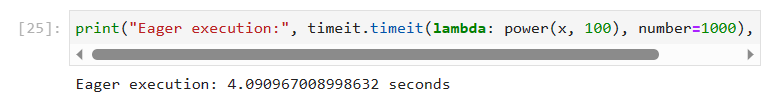
  result = tf.eye(10, dtype=tf.dtypes.int32)

  for \_ in range(y):

    result = tf.matmul(x, result)

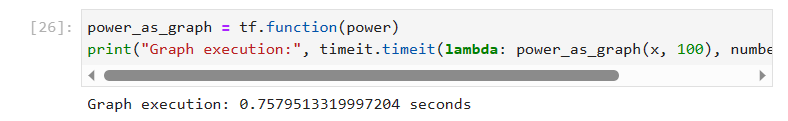
  return result

print("Eager execution:", timeit.timeit(lambda: power(x, 100), number=1000), "seconds")



power\_as\_graph = tf.function(power)

print("Graph execution:", timeit.timeit(lambda: power\_as\_graph(x, 100), number=1000), "seconds")



**7.Function 何时进行跟踪？**

要确定您的 Function 何时进行跟踪，请在其代码中添加一条 print 语句。根据经验，Function 将在每次跟踪时执行该 print 语句。

@tf.function

def a\_function\_with\_python\_side\_effect(x):

  print("Tracing!") # An eager-only side effect.

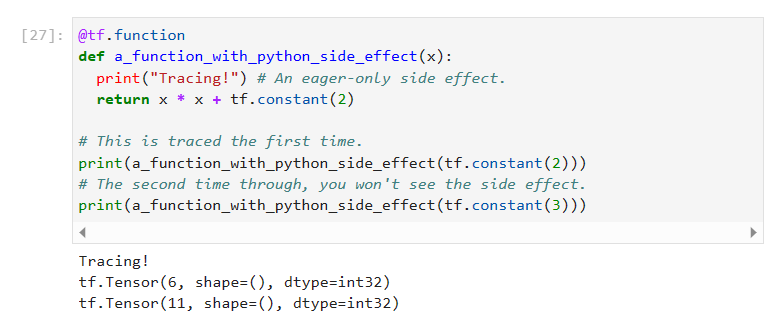
  return x \* x + tf.constant(2)

# This is traced the first time.

print(a\_function\_with\_python\_side\_effect(tf.constant(2)))

# The second time through, you won't see the side effect.

print(a\_function\_with\_python\_side\_effect(tf.constant(3)))



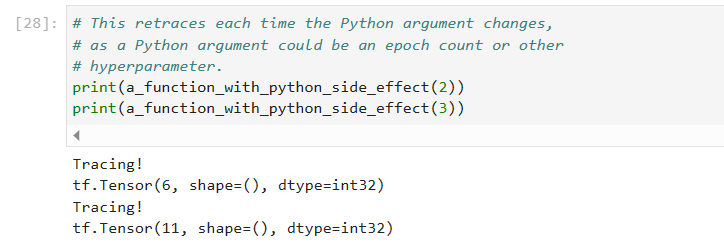
# This retraces each time the Python argument changes,

# as a Python argument could be an epoch count or other

# hyperparameter.

print(a\_function\_with\_python\_side\_effect(2))

print(a\_function\_with\_python\_side\_effect(3))



新的 Python 参数总是会触发新计算图的创建，因此需要额外的跟踪。

**（五）模块、层和模型简介**

**1.设置**

import tensorflow as tf

from datetime import datetime

%load\_ext tensorboard

**2. 在 TensorFlow 中定义模型和层**

大多数模型都由层组成。层是具有已知数学结构的函数，可以重复使用并具有可训练的变量。在 TensorFlow 中，层和模型的大多数高级实现（例如 Keras 或 Sonnet）都在以下同一个基础类上构建：tf.Module。

下面是一个在标量张量上运行的非常简单的 tf.Module 示例：

class SimpleModule(tf.Module):

  def \_\_init\_\_(self, name=None):

    super().\_\_init\_\_(name=name)

    self.a\_variable = tf.Variable(5.0, name="train\_me")

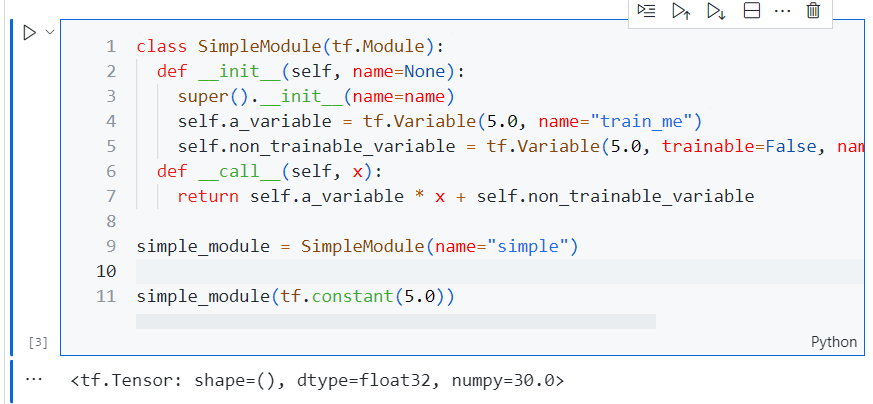
    self.non\_trainable\_variable = tf.Variable(5.0, trainable=False, name="do\_not\_train\_me")

  def \_\_call\_\_(self, x):

    return self.a\_variable \* x + self.non\_trainable\_variable

simple\_module = SimpleModule(name="simple")

simple\_module(tf.constant(5.0))



模块和引申而来的层是“对象”的深度学习术语：它们具有内部状态以及使用该状态的方法。

\_\_call\_\_ 并无特殊之处，只是其行为与 Python 可调用对象类似；您可以使用任何函数来调用模型。

您可以出于任何原因开启和关闭变量的可训练性，包括在微调过程中冻结层和变量。

注：tf.Module 是 tf.keras.layers.Layer 和 tf.keras.Model 的基类，因此您在此处看到的一切内容也适用于 Keras。出于历史兼容性原因，Keras 层不会从模块收集变量，因此您的模型应仅使用模块或仅使用 Keras 层。不过，下面给出的用于检查变量的方法在这两种情况下相同。

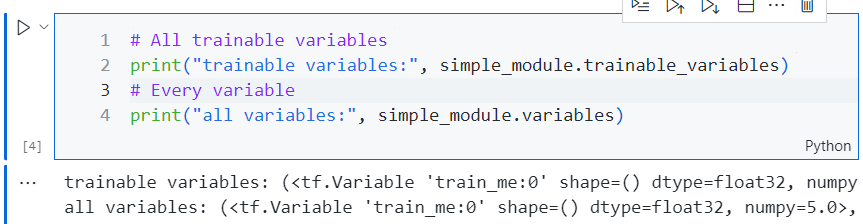
通过将 tf.Module 子类化，将自动收集分配给该对象属性的任何 tf.Variable 或 tf.Module 实例。这样，您可以保存和加载变量，还可以创建 tf.Module 的集合。

# All trainable variables

print("trainable variables:", simple\_module.trainable\_variables)

# Every variable

print("all variables:", simple\_module.variables)



下面是一个由模块组成的两层线性层模型的示例。

首先是一个密集（线性）层：

class Dense(tf.Module):

  def \_\_init\_\_(self, in\_features, out\_features, name=None):

    super().\_\_init\_\_(name=name)

    self.w = tf.Variable(

      tf.random.normal([in\_features, out\_features]), name='w')

    self.b = tf.Variable(tf.zeros([out\_features]), name='b')

  def \_\_call\_\_(self, x):

    y = tf.matmul(x, self.w) + self.b

    return tf.nn.relu(y)

随后是完整的模型，此模型将创建并应用两个层实例：

class SequentialModule(tf.Module):

  def \_\_init\_\_(self, name=None):

    super().\_\_init\_\_(name=name)

    self.dense\_1 = Dense(in\_features=3, out\_features=3)

    self.dense\_2 = Dense(in\_features=3, out\_features=2)

  def \_\_call\_\_(self, x):

    x = self.dense\_1(x)

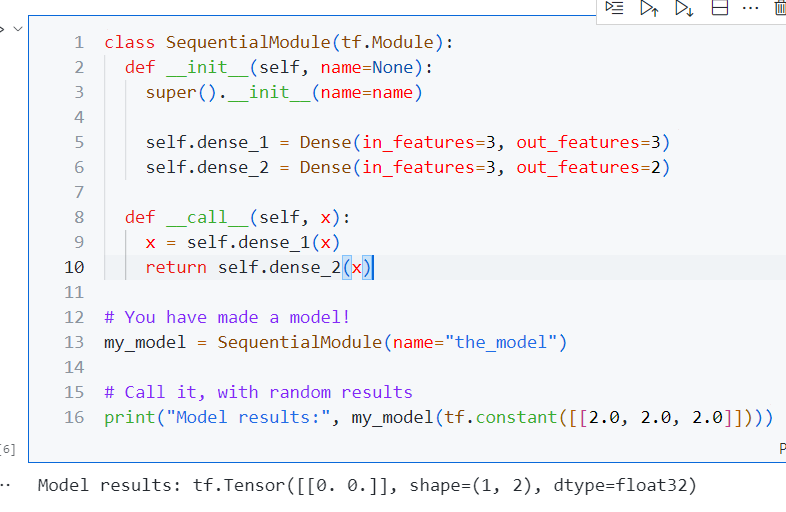
    return self.dense\_2(x)

# You have made a model!

my\_model = SequentialModule(name="the\_model")

# Call it, with random results

print("Model results:", my\_model(tf.constant([[2.0, 2.0, 2.0]])))

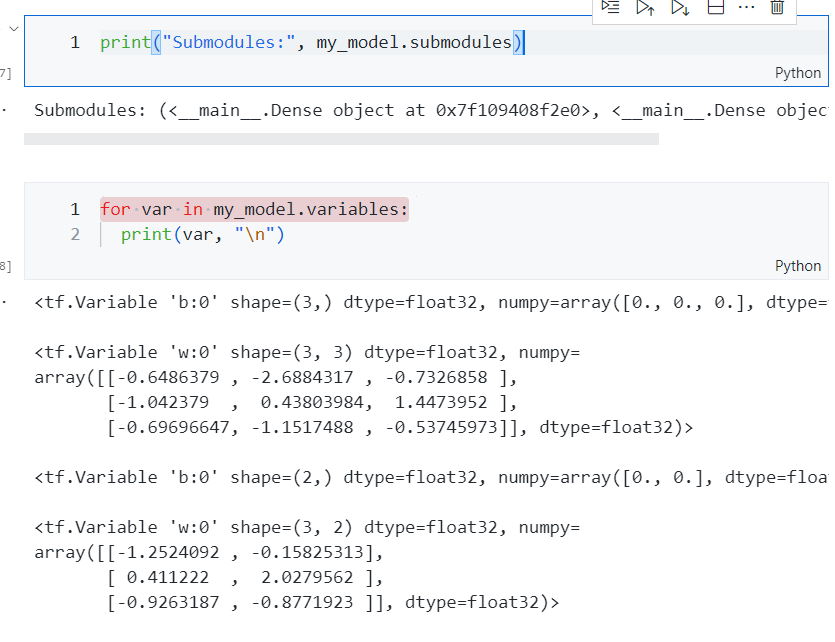


tf.Module 实例将以递归方式自动收集分配给它的任何 tf.Variable 或 tf.Module 实例。这样，您可以使用单个模型实例管理 tf.Module 的集合，并保存和加载整个模型。

print("Submodules:", my\_model.submodules)

for var in my\_model.variables:

  print(var, "\n")



**3. 等待创建变量**

您在这里可能已经注意到，必须定义层的输入和输出大小。这样，w 变量才会具有已知的形状并且可被分配。

通过将变量创建推迟到第一次使用特定输入形状调用模块时，您将无需预先指定输入大小。

class FlexibleDenseModule(tf.Module):

  # Note: No need for `in\_features`

  def \_\_init\_\_(self, out\_features, name=None):

    super().\_\_init\_\_(name=name)

    self.is\_built = False

    self.out\_features = out\_features

  def \_\_call\_\_(self, x):

    # Create variables on first call.

    if not self.is\_built:

      self.w = tf.Variable(

        tf.random.normal([x.shape[-1], self.out\_features]), name='w')

      self.b = tf.Variable(tf.zeros([self.out\_features]), name='b')

      self.is\_built = True

    y = tf.matmul(x, self.w) + self.b

    return tf.nn.relu(y)

# Used in a module

class MySequentialModule(tf.Module):

  def \_\_init\_\_(self, name=None):

    super().\_\_init\_\_(name=name)

    self.dense\_1 = FlexibleDenseModule(out\_features=3)

    self.dense\_2 = FlexibleDenseModule(out\_features=2)

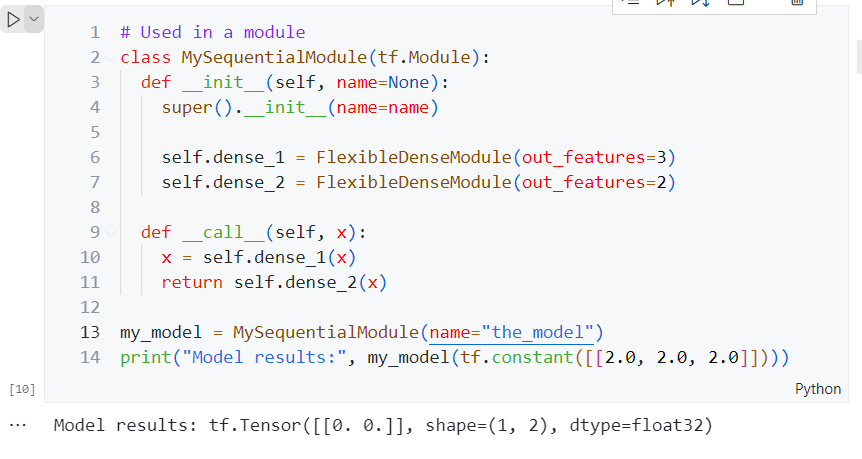
  def \_\_call\_\_(self, x):

    x = self.dense\_1(x)

    return self.dense\_2(x)

my\_model = MySequentialModule(name="the\_model")

print("Model results:", my\_model(tf.constant([[2.0, 2.0, 2.0]])))



这种灵活性是 TensorFlow 层通常仅需要指定其输出的形状（例如在 tf.keras.layers.Dense 中），而无需指定输入和输出大小的原因。

**4.保存权重**

您可以将 tf.Module 保存为检查点和 SavedModel。

检查点即是权重（即模块及其子模块内部的变量集的值）。

chkp\_path = "my\_checkpoint"

checkpoint = tf.train.Checkpoint(model=my\_model)

checkpoint.write(chkp\_path)



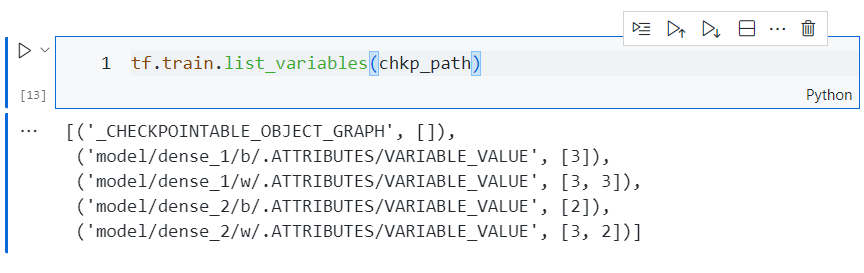
检查点由两种文件组成---数据本身以及元数据的索引文件。索引文件跟踪实际保存的内容和检查点的编号，而检查点数据包含变量值及其特性查找路径。

*!*ls my\_checkpoint\*



您可以查看检查点内部，以确保整个变量集合已由包含这些变量的 Python 对象保存并排序。

tf.train.list\_variables(chkp\_path)



在分布式（多机）训练期间，可以将它们分片，这就是要对它们进行编号（例如 '00000-of-00001'）的原因。不过，在本例中，只有一个分片。重新加载模型时，将重写 Python 对象中的值。

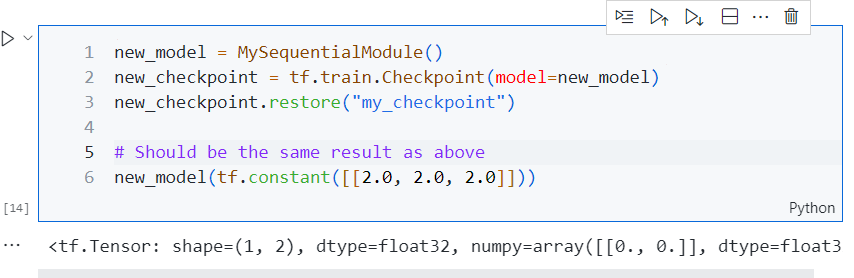
new\_model = MySequentialModule()

new\_checkpoint = tf.train.Checkpoint(model=new\_model)

new\_checkpoint.restore("my\_checkpoint")

# Should be the same result as above

new\_model(tf.constant([[2.0, 2.0, 2.0]]))



**5.保存函数**

TensorFlow 可以在不使用原始 Python 对象的情况下运行模型，如 TensorFlow Serving 和 TensorFlow Lite 所示，甚至当您从 TensorFlow Hub 下载经过训练的模型时也是如此。

TensorFlow 需要了解如何执行 Python 中描述的计算，但不需要原始代码。为此，您可以创建一个计算图，如计算图和函数简介指南中所述。

此计算图中包含实现函数的运算。

您可以通过添加 @tf.function 装饰器在上面的模型中定义计算图，以指示此代码应作为计算图运行。

class MySequentialModule(tf.Module):

  def \_\_init\_\_(self, name=None):

    super().\_\_init\_\_(name=name)

    self.dense\_1 = Dense(in\_features=3, out\_features=3)

    self.dense\_2 = Dense(in\_features=3, out\_features=2)

  @tf.function

  def \_\_call\_\_(self, x):

    x = self.dense\_1(x)

    return self.dense\_2(x)

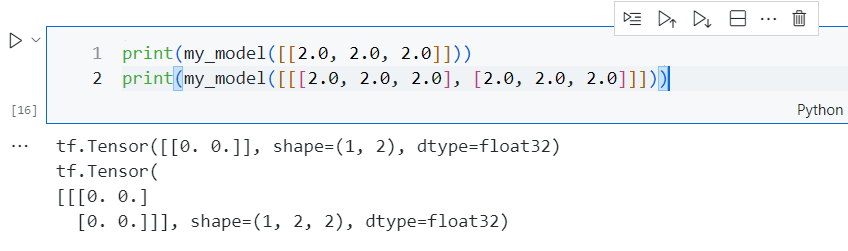
# You have made a model with a graph!

my\_model = MySequentialModule(name="the\_model")

您构建的模块的工作原理与之前完全相同。传递给函数的每个唯一签名都会创建一个单独的计算图。请参阅计算图和函数简介指南以了解详情。

print(my\_model([[2.0, 2.0, 2.0]]))

print(my\_model([[[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]]))



您可以通过在 TensorBoard 摘要中跟踪计算图来将其可视化。

# Set up logging.

stamp = datetime.now().strftime("%Y%m%d-%H%M%S")

logdir = "logs/func/%s" % stamp

writer = tf.summary.create\_file\_writer(logdir)

# Create a new model to get a fresh trace

# Otherwise the summary will not see the graph.

new\_model = MySequentialModule()

# Bracket the function call with

# tf.summary.trace\_on() and tf.summary.trace\_export().

tf.summary.trace\_on(graph=True)

tf.profiler.experimental.start(logdir)

# Call only one tf.function when tracing.

z = print(new\_model(tf.constant([[2.0, 2.0, 2.0]])))

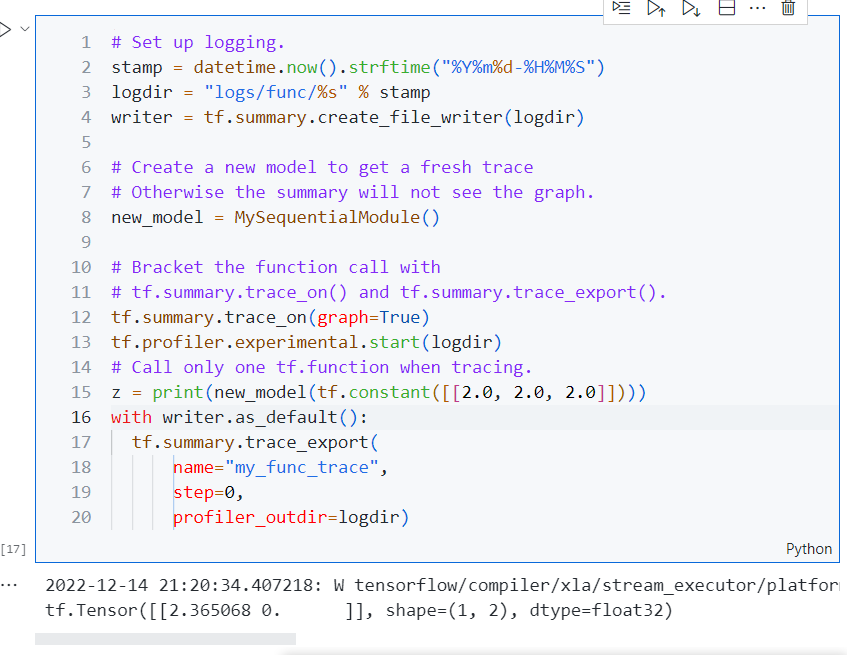
with writer.as\_default():

  tf.summary.trace\_export(

      name="my\_func\_trace",

      step=0,

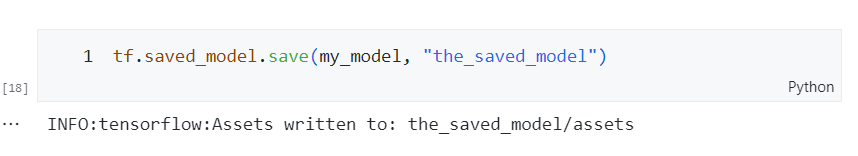
      profiler\_outdir=logdir)



**6.创建SavedModel**

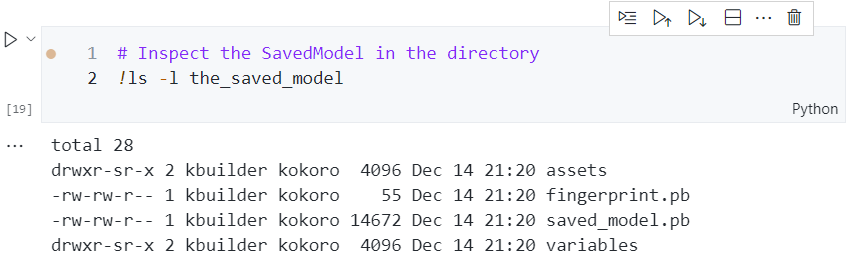
共享经过完全训练的模型的推荐方式是使用 SavedModel。SavedModel 包含函数集合与权重集合。您可以按以下方式保存刚刚训练的模型：

tf.saved\_model.save(my\_model, "the\_saved\_model")



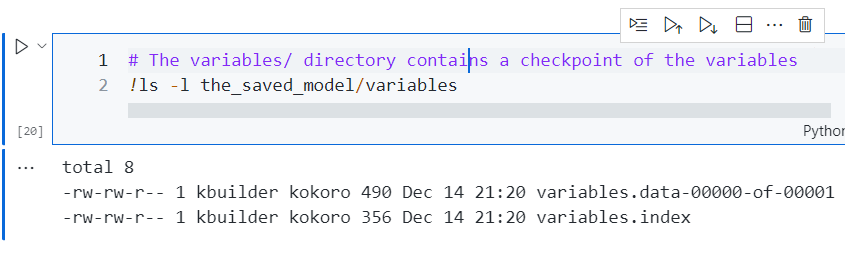
# Inspect the SavedModel in the directory

*!*ls -l the\_saved\_model



# The variables/ directory contains a checkpoint of the variables

*!*ls -l the\_saved\_model/variables



saved\_model.pb 文件是一个描述函数式 tf.Graph 的协议缓冲区。

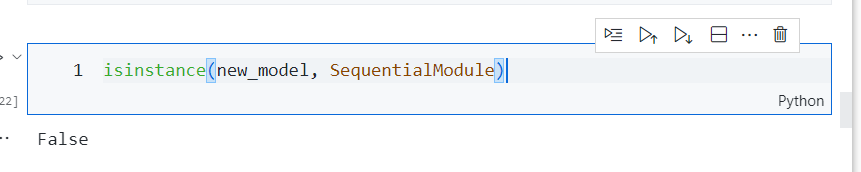
可以从此表示加载模型和层，而无需实际构建创建该表示的类的实例。在您没有（或不需要）Python 解释器（例如大规模应用或在边缘设备上），或者在原始 Python 代码不可用或不实用的情况下，这样做十分理想。

您可以将模型作为新对象加载：

new\_model = tf.saved\_model.load("the\_saved\_model")

通过加载已保存模型创建的 `new\_model` 是 TensorFlow 内部的用户对象，无需任何类知识。它不是 `SequentialModule` 类型的对象。

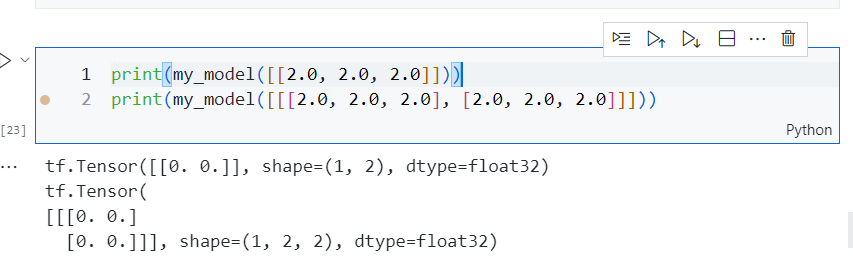
isinstance(new\_model, SequentialModule)



此新模型​​适用于已定义的输入签名。您不能向以这种方式恢复的模型添加更多签名。

print(my\_model([[2.0, 2.0, 2.0]]))

print(my\_model([[[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]]))



因此，利用 `SavedModel`，您可以使用 `tf.Module` 保存 TensorFlow 权重和计算图，随后再次加载它们。

**7.** **Keras 层**

tf.keras.layers.Layer 是所有 Keras 层的基类，它继承自 tf.Module。

您只需换出父项，然后将 \_\_call\_\_ 更改为 call 即可将模块转换为 Keras 层：

class MyDense(tf.keras.layers.Layer):

  # Adding \*\*kwargs to support base Keras layer arguments

  def \_\_init\_\_(self, in\_features, out\_features, \*\*kwargs):

    super().\_\_init\_\_(\*\*kwargs)

    # This will soon move to the build step; see below

    self.w = tf.Variable(

      tf.random.normal([in\_features, out\_features]), name='w')

    self.b = tf.Variable(tf.zeros([out\_features]), name='b')

  def call(self, x):

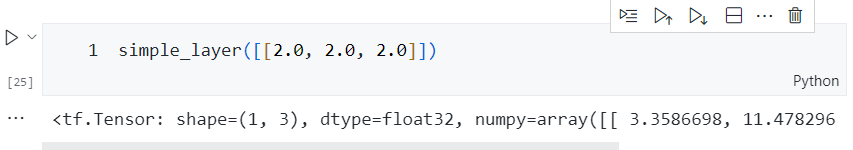
    y = tf.matmul(x, self.w) + self.b

    return tf.nn.relu(y)

simple\_layer = MyDense(name="simple", in\_features=3, out\_features=3)

Keras 层有自己的 `\_\_call\_\_`，它会进行下一部分中所述的某些簿记，然后调用 `call()`。您应当不会看到功能上的任何变化。

simple\_layer([[2.0, 2.0, 2.0]])



**8.build 步骤**

如上所述，在您确定输入形状之前，等待创建变量在许多情况下十分方便。

Keras 层具有额外的生命周期步骤，可让您在定义层时获得更高的灵活性。这是在 build() 函数中定义的。

build 仅被调用一次，而且是使用输入的形状调用的。它通常用于创建变量（权重）。

您可以根据输入的大小灵活地重写上面的 MyDense 层:

class FlexibleDense(tf.keras.layers.Layer):

  # Note the added `\*\*kwargs`, as Keras supports many arguments

  def \_\_init\_\_(self, out\_features, \*\*kwargs):

    super().\_\_init\_\_(\*\*kwargs)

    self.out\_features = out\_features

  def build(self, input\_shape):  # Create the state of the layer (weights)

    self.w = tf.Variable(

      tf.random.normal([input\_shape[-1], self.out\_features]), name='w')

    self.b = tf.Variable(tf.zeros([self.out\_features]), name='b')

  def call(self, inputs):  # Defines the computation from inputs to outputs

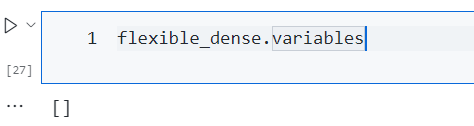
    return tf.matmul(inputs, self.w) + self.b

# Create the instance of the layer

flexible\_dense = FlexibleDense(out\_features=3)

此时，模型尚未构建，因此没有变量:

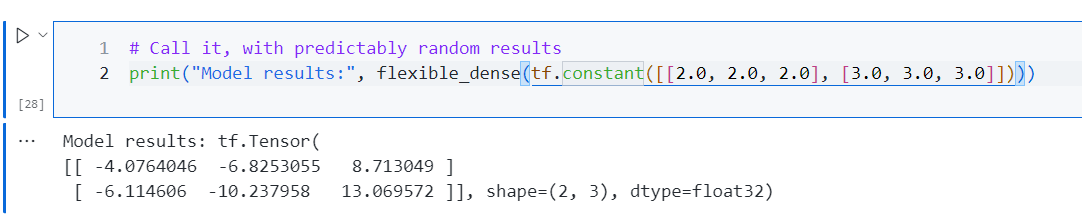
flexible\_dense.variables



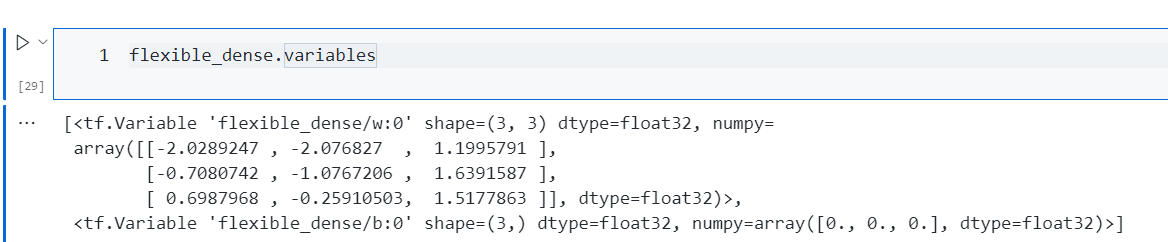
调用该函数会分配大小适当的变量。

# Call it, with predictably random results

print("Model results:", flexible\_dense(tf.constant([[2.0, 2.0, 2.0], [3.0, 3.0, 3.0]])))



flexible\_dense.variables



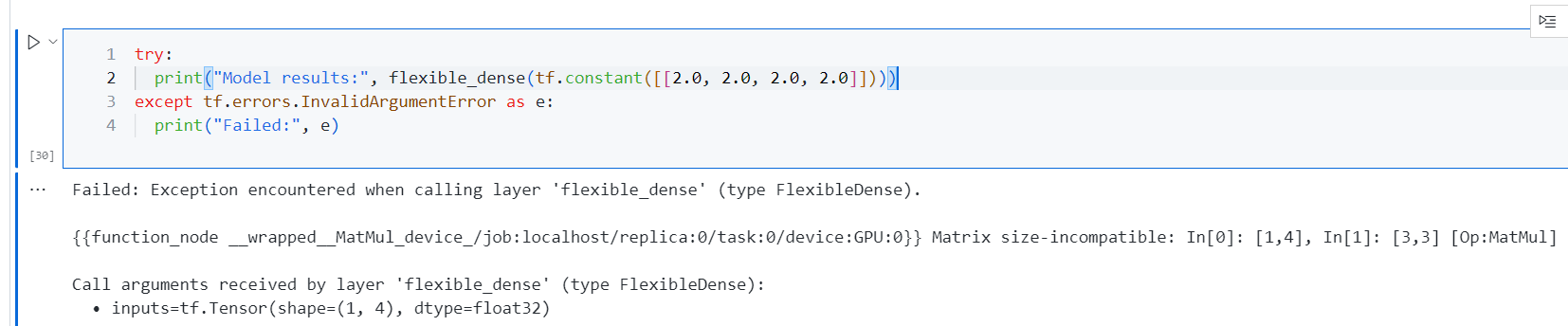
由于仅调用一次 `build`，因此如果输入形状与层的变量不兼容，输入将被拒绝。

try:

  print("Model results:", flexible\_dense(tf.constant([[2.0, 2.0, 2.0, 2.0]])))

except tf.errors.InvalidArgumentError as e:

  print("Failed:", e)



Keras 层具有许多额外的功能，包括：

可选损失、对指标的支持、对可选 training 参数的内置支持，用于区分训练和推断用途

get\_config 和 from\_config 方法，允许您准确存储配置以在 Python 中克隆模型

**9.Keras 模型**

您可以将模型定义为嵌套的 Keras 层。

但是，Keras 还提供了称为 tf.keras.Model 的全功能模型类。它继承自 tf.keras.layers.Layer，因此 Keras 模型支持以同样的方式使用、嵌套和保存。Keras 模型还具有额外的功能，这使它们可以轻松训练、评估、加载、保存，甚至在多台机器上进行训练。

您可以使用几乎相同的代码定义上面的 SequentialModule，再次将 \_\_call\_\_ 转换为 call() 并更改父项。

class MySequentialModel(tf.keras.Model):

  def \_\_init\_\_(self, name=None, \*\*kwargs):

    super().\_\_init\_\_(\*\*kwargs)

    self.dense\_1 = FlexibleDense(out\_features=3)

    self.dense\_2 = FlexibleDense(out\_features=2)

  def call(self, x):

    x = self.dense\_1(x)

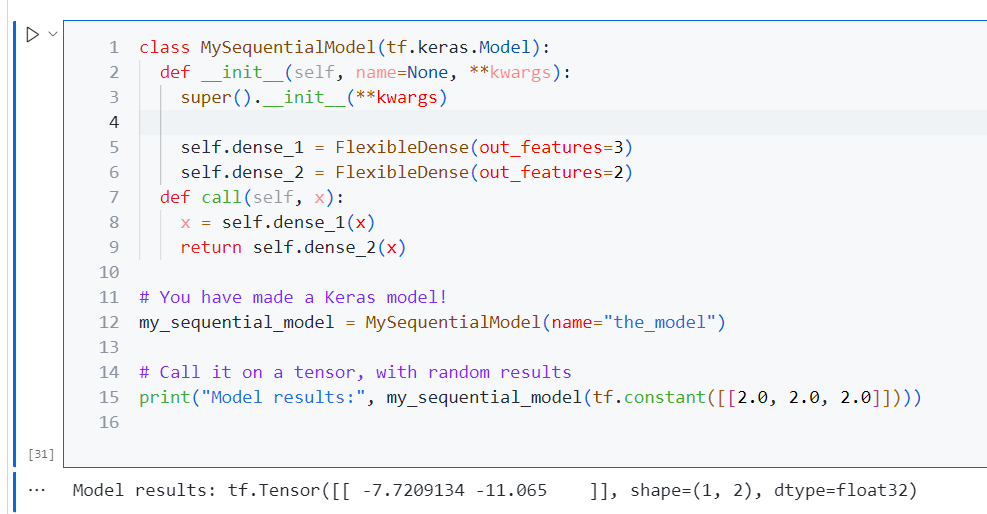
    return self.dense\_2(x)

# You have made a Keras model!

my\_sequential\_model = MySequentialModel(name="the\_model")

# Call it on a tensor, with random results

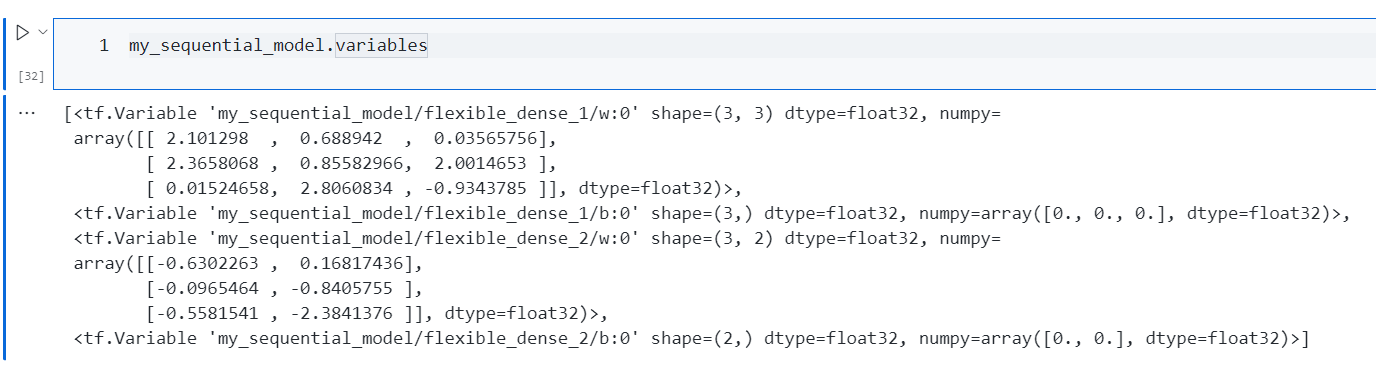
print("Model results:", my\_sequential\_model(tf.constant([[2.0, 2.0, 2.0]])))



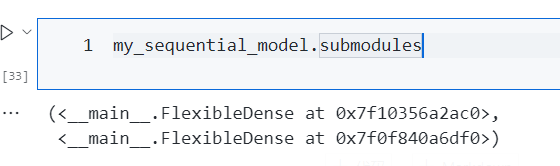
所有相同的功能都可用，包括跟踪变量和子模块。

注：为了强调上面的注意事项，嵌套在 Keras 层或模型中的原始 `tf.Module` 将不会收集其变量以用于训练或保存。相反，它会在 Keras 层内嵌套 Keras 层。

my\_sequential\_model.variables



my\_sequential\_model.submodules



重写 `tf.keras.Model` 是一种构建 TensorFlow 模型的极 Python 化方式。如果要从其他框架迁移模型，这可能非常简单。如果要构造的模型是现有层和输入的简单组合，则可以使用[函数式 API](./keras/functional.ipynb) 节省时间和空间，此 API 附带有关模型重构和架构的附加功能。

下面是使用函数式 API 构造的相同模型：

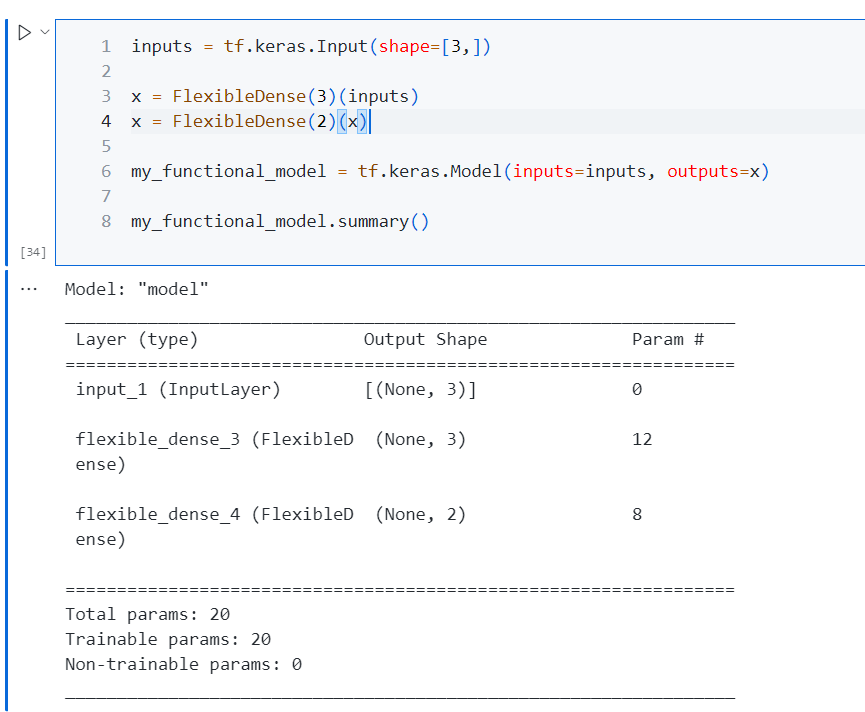
inputs = tf.keras.Input(shape=[3,])

x = FlexibleDense(3)(inputs)

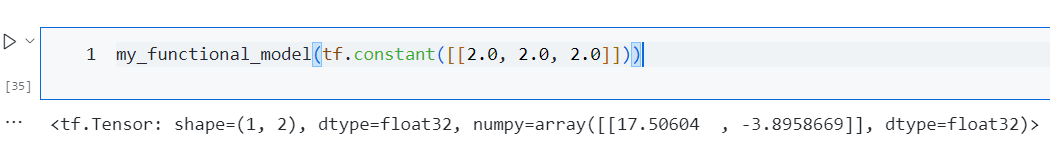
x = FlexibleDense(2)(x)

my\_functional\_model = tf.keras.Model(inputs=inputs, outputs=x)

my\_functional\_model.summary()



my\_functional\_model(tf.constant([[2.0, 2.0, 2.0]]))



这里的主要区别在于，输入形状是作为函数构造过程的一部分预先指定的。在这种情况下，不必完全指定 `input\_shape` 参数；您可以将某些维度保留为 `None`。

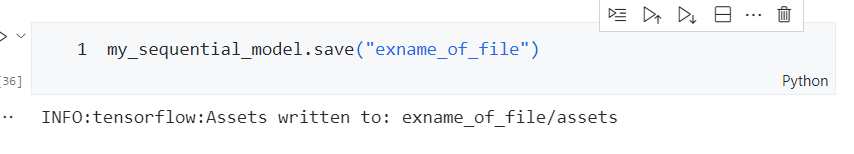
注：您无需在子类化模型中指定 `input\_shape` 或 `InputLayer`；这些参数和层将被忽略。

**10. 保存 Keras 模型**

可以为 Keras 模型创建检查点，这看起来和 tf.Module 一样。

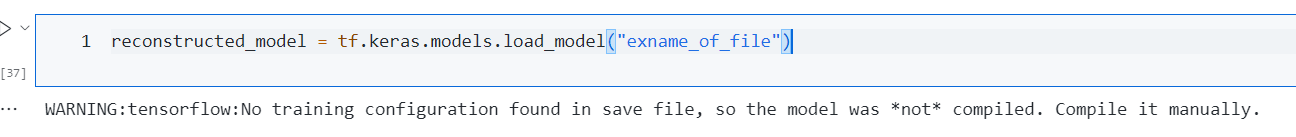
Keras 模型也可以使用 tf.saved\_models.save() 保存，因为它们是模块。但是，Keras 模型具有更方便的方法和其他功能：

my\_sequential\_model.save("exname\_of\_file")



同样地，它们也可以轻松重新加载：

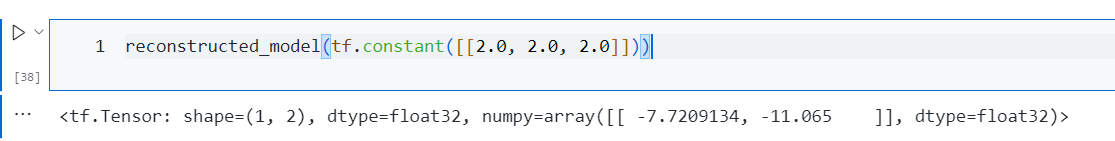
reconstructed\_model = tf.keras.models.load\_model("exname\_of\_file")



Keras `SavedModels` 还可以保存指标、损失和优化器状态。

可以使用此重构模型，并且在相同数据上调用时会产生相同的结果：

reconstructed\_model(tf.constant([[2.0, 2.0, 2.0]]))



**（六）训练循环**

**1.创建**

import tensorflow as tf

import matplotlib.pyplot as plt

colors = plt.rcParams['axes.prop\_cycle'].by\_key()['color']

**2.解决机器学习问题**

解决一个机器学习问题通常包含以下步骤：

· 获得训练数据。

· 定义模型。

· 定义损失函数。

· 遍历训练数据，从目标值计算损失。

· 计算该损失的梯度，并使用optimizer调整变量以适合数据。

· 计算结果。

为了便于说明，在本指南中，您将开发一个简单的线性模型, 其中包含两个变量:(权重)和 (偏差)。

最基本的机器学习问题：给定x和y，尝试通过简单的线性回归来找到直线的斜率和偏移量。

**3.数据**

监督学习使用输入（通常表示为 x）和输出（表示为 y，通常称为标签）。目标是从成对的输入和输出中学习，以便您可以根据输入预测输出的值。

TensorFlow中几乎每个输入数据都是由张量表示，并且通常是向量。监督学习中，输出(即想到预测值)同样是个张量。

下面是通过将高斯（正态）噪声添加到直线上的点而合成的一些数据。

# The actual line

TRUE\_W = 3.0

TRUE\_B = 2.0

NUM\_EXAMPLES = 201

# A vector of random x values

x = tf.linspace(-2,2, NUM\_EXAMPLES)

x = tf.cast(x, tf.float32)

def f(x):

  return x \* TRUE\_W + TRUE\_B

# Generate some noise

noise = tf.random.normal(shape=[NUM\_EXAMPLES])

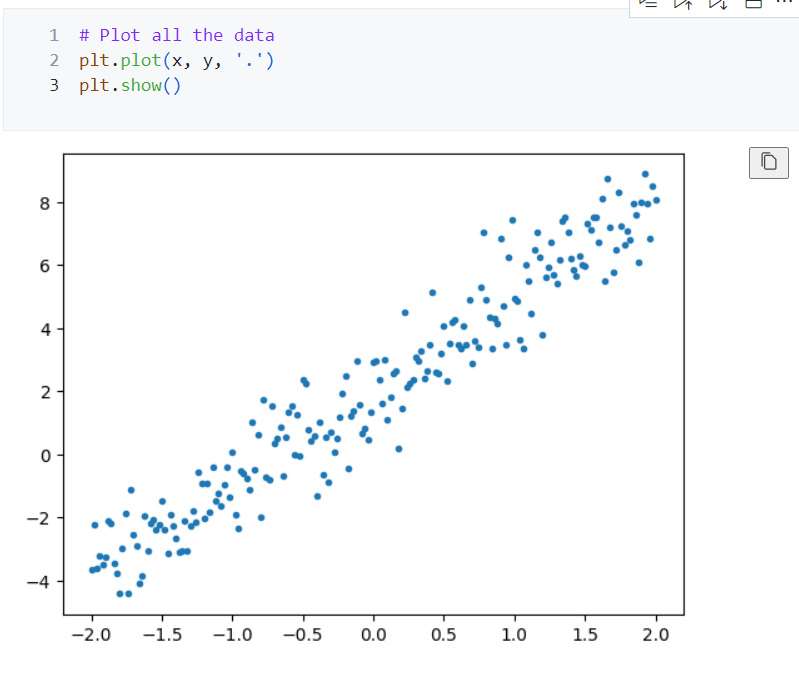
# Calculate y

y = f(x) + noise

# Plot all the data

plt.plot(x, y, '.')

plt.show()



张量通常以 \*batches\* 的形式聚集在一起，或者是成组的输入和输出堆叠在一起。批处理能够对训练过程带来一些好处，并且可以与加速器和矢量化计算很好地配合使用。给定此数据集的大小，您可以将整个数据集视为一个批次。

**4.定义模型**

使用 tf.Variable 代表模型中的所有权重。tf.Variable 能够存储值，并根据需要以张量形式提供它。详情请见 variable guide。使用 tf.Module 封装变量和计算。您可以使用任何Python对象，但是通过这种方式可以轻松保存它。

这里，您可以定义 w 和 b 为变量。

class MyModel(tf.Module):

  def \_\_init\_\_(self, \*\*kwargs):

    super().\_\_init\_\_(\*\*kwargs)

    # Initialize the weights to `5.0` and the bias to `0.0`

    # In practice, these should be randomly initialized

    self.w = tf.Variable(5.0)

    self.b = tf.Variable(0.0)

  def \_\_call\_\_(self, x):

    return self.w \* x + self.b

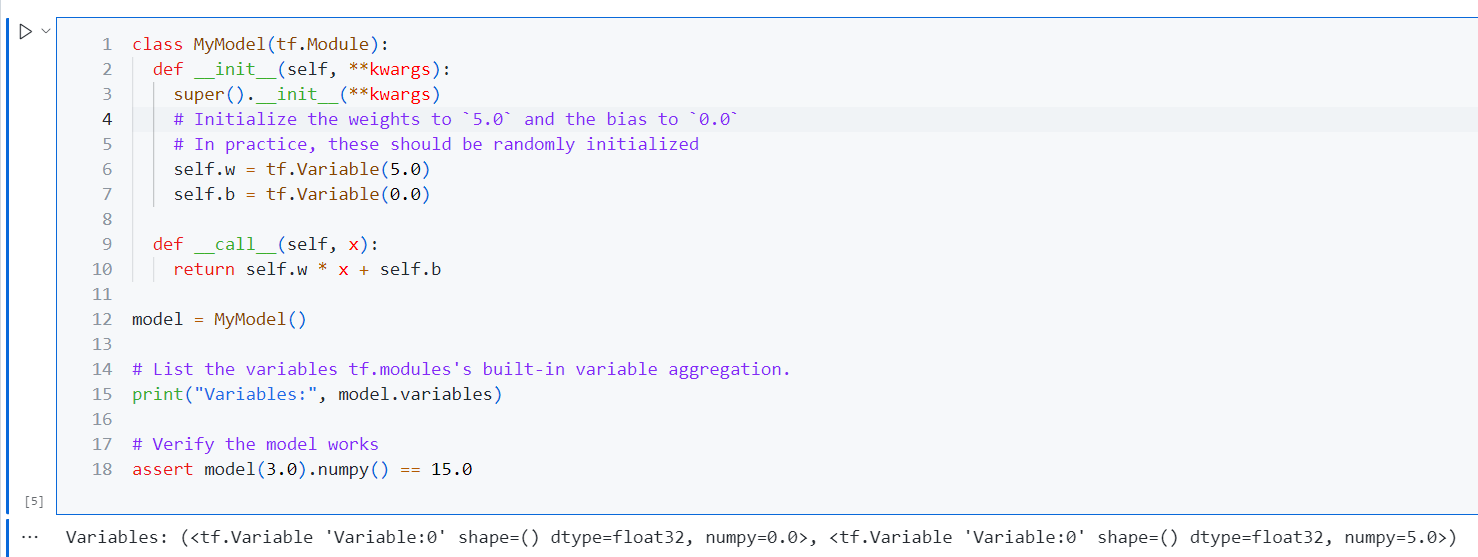
model = MyModel()

# List the variables tf.modules's built-in variable aggregation.

print("Variables:", model.variables)

# Verify the model works

assert model(3.0).numpy() == 15.0



初始变量在此处以固定方式设置，但 Keras 提供了您可以与或不与 Keras 其他部分一起使用的许多初始值设定项

**5.定义损失函数**

损失函数衡量给定输入的模型输出与目标输出的匹配程度。目的是在训练过程中尽量减少这种差异。定义标准的L2损失，也称为“均方误差”:

# This computes a single loss value for an entire batch

def loss(target\_y, predicted\_y):

  return tf.reduce\_mean(tf.square(target\_y - predicted\_y))

在训练模型之前，您可以可视化损失值。使用红色绘制模型的预测值，使用蓝色绘制训练数据。

plt.plot(x, y, '.', label="Data")

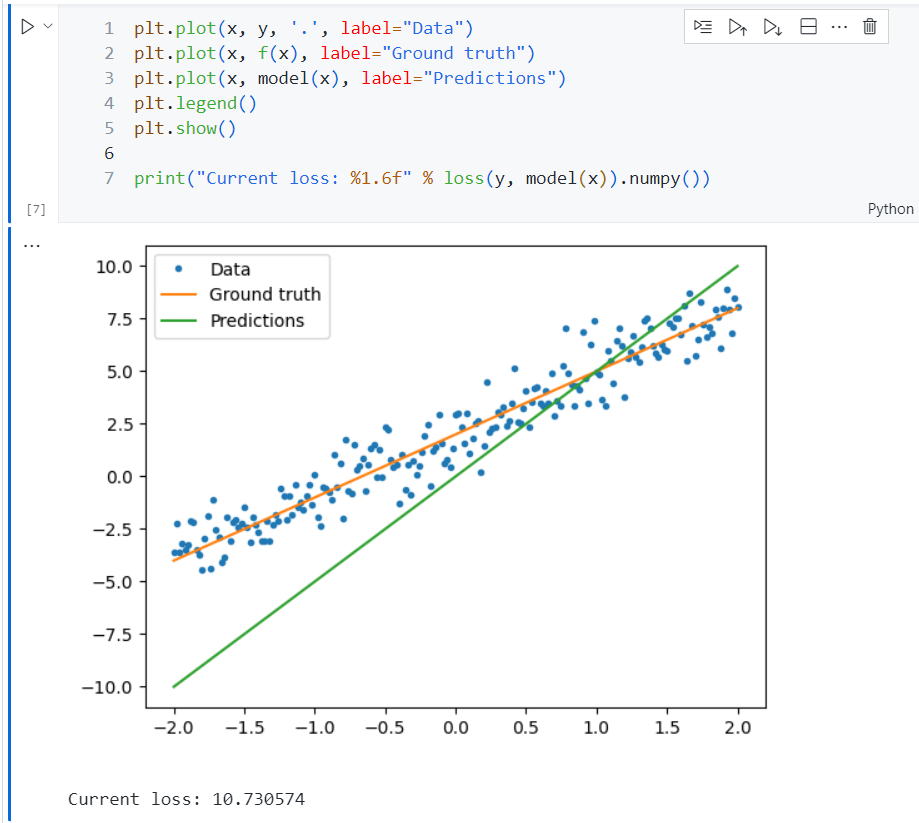
plt.plot(x, f(x), label="Ground truth")

plt.plot(x, model(x), label="Predictions")

plt.legend()

plt.show()

print("Current loss: %1.6f" % loss(y, model(x)).numpy())



**6.定义训练循环**

训练循环按顺序重复执行以下任务：

· 发送一批输入值，通过模型生成输出值

· 通过比较输出值与输出（标签），来计算损失值

· 使用梯度带(GradientTape)找到梯度值

· 使用这些梯度优化变量

这个例子中，您可以使用 gradient descent训练数据。

tf.keras.optimizers中有许多梯度下降的变量。但是本着搭建的第一原则，您将在这里 借助tf.GradientTape的自动微分和tf.assign\_sub的递减值（结合了tf.assign和tf.sub）自己实现基本数学：

# Given a callable model, inputs, outputs, and a learning rate...

def train(model, x, y, learning\_rate):

  with tf.GradientTape() as t:

    # Trainable variables are automatically tracked by GradientTape

    current\_loss = loss(y, model(x))

  # Use GradientTape to calculate the gradients with respect to W and b

  dw, db = t.gradient(current\_loss, [model.w, model.b])

  # Subtract the gradient scaled by the learning rate

  model.w.assign\_sub(learning\_rate \* dw)

  model.b.assign\_sub(learning\_rate \* db)

要查看训练，您可以通过训练循环发送同一批次的 \*x\* 和 \*y\*，并观察 `W` 和 `b` 如何变化。

model = MyModel()

# Collect the history of W-values and b-values to plot later

weights = []

biases = []

epochs = range(10)

# Define a training loop

def report(model, loss):

  return f"W = {model.w.numpy():1.2f}, b = {model.b.numpy():1.2f}, loss={loss:2.5f}"

def training\_loop(model, x, y):

  for epoch in epochs:

    # Update the model with the single giant batch

    train(model, x, y, learning\_rate=0.1)

    # Track this before I update

    weights.append(model.w.numpy())

    biases.append(model.b.numpy())

    current\_loss = loss(y, model(x))

    print(f"Epoch {epoch:2d}:")

    print("    ", report(model, current\_loss))

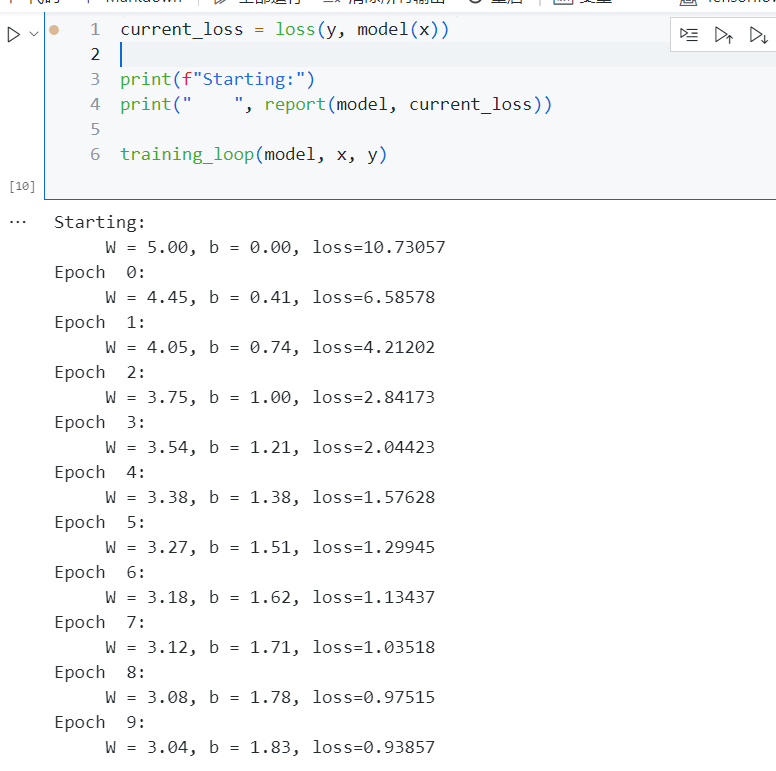
进行训练

current\_loss = loss(y, model(x))

print(f"Starting:")

print("    ", report(model, current\_loss))

training\_loop(model, x, y)



下面是权重随时间的演变：

plt.plot(epochs, weights, label='Weights', color=colors[0])

plt.plot(epochs, [TRUE\_W] \* len(epochs), '--',

         label = "True weight", color=colors[0])

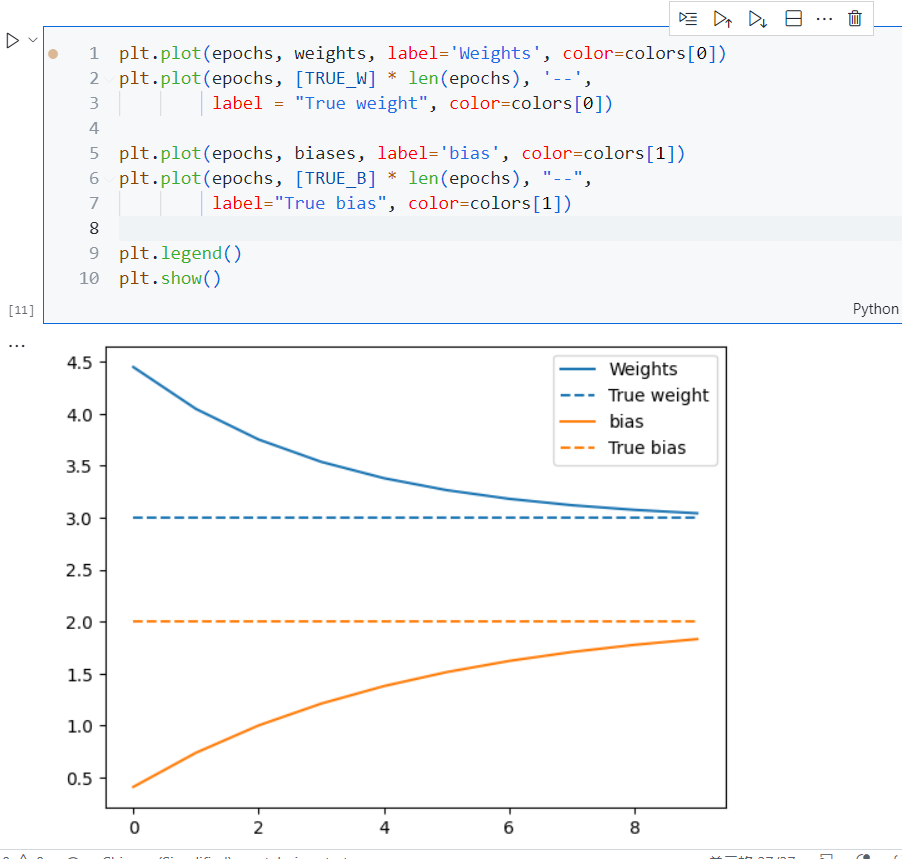
plt.plot(epochs, biases, label='bias', color=colors[1])

plt.plot(epochs, [TRUE\_B] \* len(epochs), "--",

         label="True bias", color=colors[1])

plt.legend()

plt.show()



呈现训练的模型的性能

plt.plot(x, y, '.', label="Data")

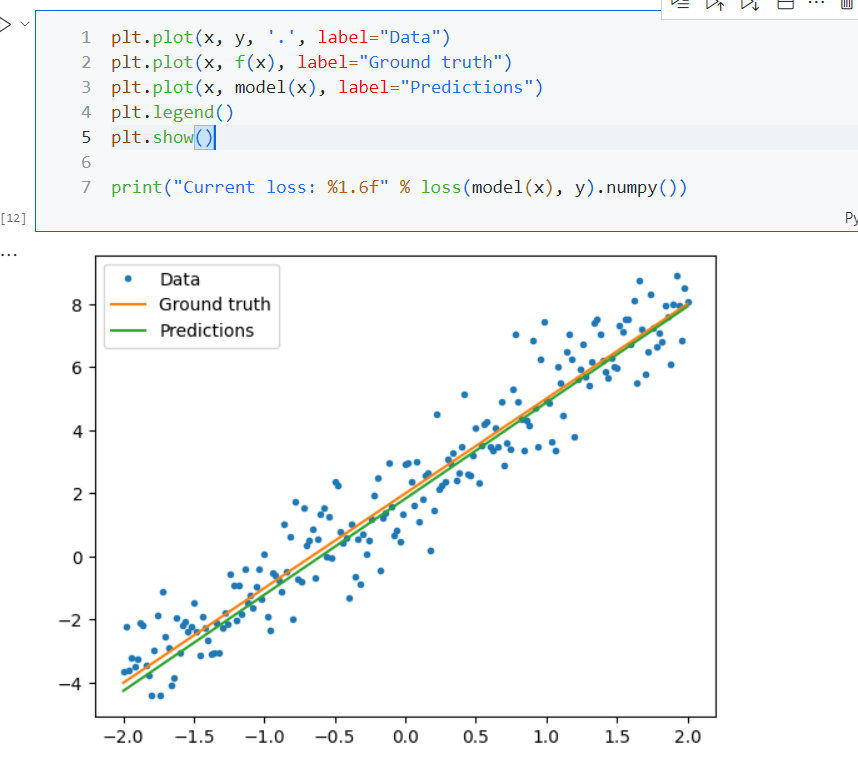
plt.plot(x, f(x), label="Ground truth")

plt.plot(x, model(x), label="Predictions")

plt.legend()

plt.show()

print("Current loss: %1.6f" % loss(model(x), y).numpy())



**7. 使用Keras完成相同的解决方案**

将上面的代码与Keras中的等效代码进行对比很有用。

如果您将tf.keras.Model子类化，则定义模型与其看起来完全相同。请记住，Keras模型最终从模块继承。

class MyModelKeras(tf.keras.Model):

  def \_\_init\_\_(self, \*\*kwargs):

    super().\_\_init\_\_(\*\*kwargs)

    # Initialize the weights to `5.0` and the bias to `0.0`

    # In practice, these should be randomly initialized

    self.w = tf.Variable(5.0)

    self.b = tf.Variable(0.0)

  def call(self, x):

    return self.w \* x + self.b

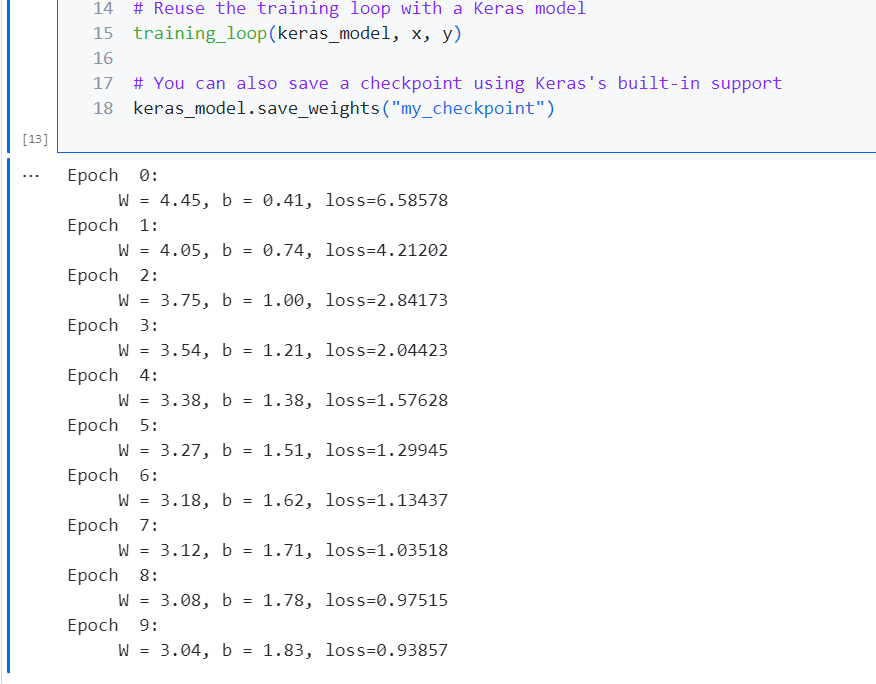
keras\_model = MyModelKeras()

# Reuse the training loop with a Keras model

training\_loop(keras\_model, x, y)

# You can also save a checkpoint using Keras's built-in support

keras\_model.save\_weights("my\_checkpoint")



您可以使用Keras的内置功能作为捷径，而不必在每次创建模型时都编写新的训练循环。当您不想编写或调试Python训练循环时，这很有用。

如果您使用Keras，您将会需要使用 `model.compile()` 去设置参数, 使用`model.fit()` 进行训练。借助Keras实现L2损失和梯度下降需要的代码量更少，就像一个捷径。Keras损失和优化器也可以在这些便利功能之外使用，而前面的示例也可以使用它们。

keras\_model = MyModelKeras()

# compile sets the training parameters

keras\_model.compile(

    # By default, fit() uses tf.function().  You can

    # turn that off for debugging, but it is on now.

    run\_eagerly=False,

    # Using a built-in optimizer, configuring as an object

    optimizer=tf.keras.optimizers.SGD(learning\_rate=0.1),

    # Keras comes with built-in MSE error

    # However, you could use the loss function

    # defined above

    loss=tf.keras.losses.mean\_squared\_error,

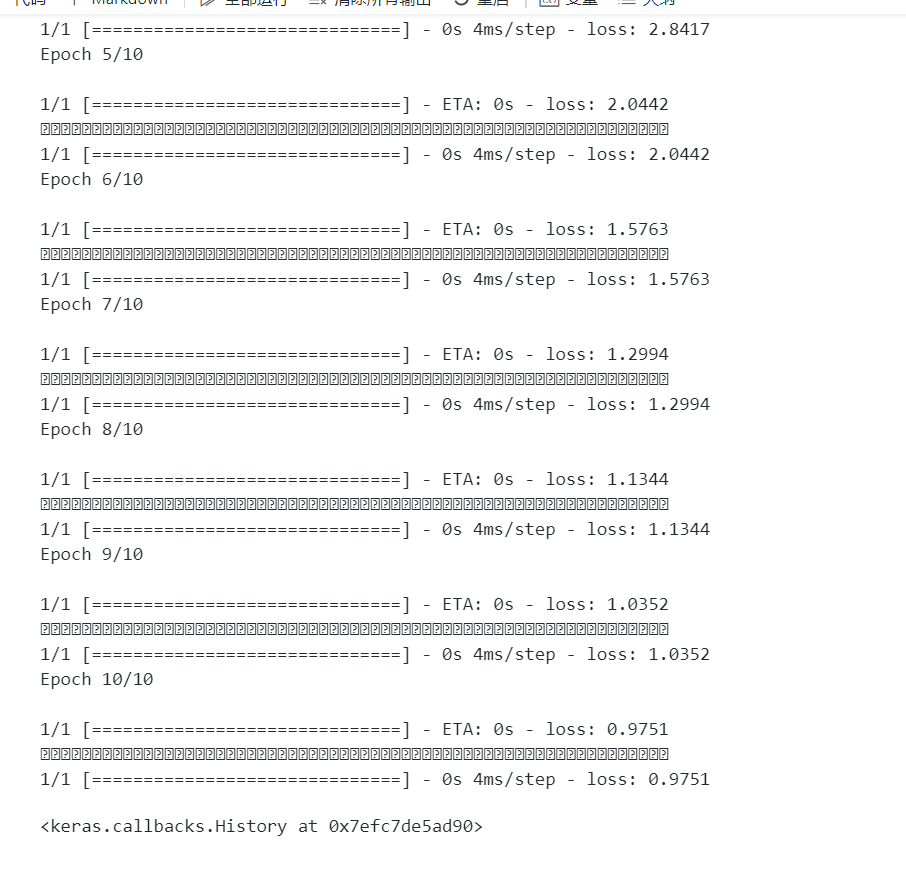
)

Keras`fit`期望批处理数据或完整的数据集作为NumPy数组。 NumPy数组分为多个批次，默认批次大小为32。

这一案例中，为了匹配手写训练循环，您应该以大小为1000的单批次传递x。

print(x.shape[0])

keras\_model.fit(x, y, epochs=10, batch\_size=1000)



Keras会在训练后而不是之前打印出损失，因此第一次损失会显得较低。否则，这表明本质上相同的训练效果。

1. **实验总结**

通过本次实验，我进一步提高了自己的深度学习基础和实践经验，并深入认识了 TensorFlow 的设计思想和工作方式，对于提升自己的计算机视觉、自然语言处理、强化学习等方面的研究能力和应用水平有着积极的作用。在实践中，我也遇到了一些问题，例如了解 TensorFlow 的版本兼容性、学习使用 TensorFlow API 和 TensorFlow 计算图的相关知识，这些问题最终都得到了解决，并让我在实践中更加熟练掌握了 TensorFlow 的基础知识和应用技巧。总体来看，本次实验让我深入了解了 TensorFlow 的计算图原理和计算图优化的实现原理，对于后续深度学习模型的理解和实现也受益匪浅。我深刻了解了 TensorFlow 的函数和图的概念，掌握了 TensorFlow 基础知识与应用。主要学习到如下知识点：

1. TensorFlow 函数可以通过 `tf.function()` 装饰器将 Python 函数转换为 TensorFlow 函数，并可以使用 `get\_concrete\_function()` 方法在计算图中生成具体的函数实例。

2. TensorFlow 可以将 Python 代码转换为 TensorFlow 图，这样可以通过 TensorFlow 的资源管理器缓存计算图来加速计算。

3. 在处理数据集时，可以用 `tf.data.Dataset()` 效率更高地处理大量数据。

4. TensorFlow 同样是允许计算浮点数简单操作的，即 TensorFlow 可以用作数值计算的计算库。

总结来说，TensorFlow 的使用让我深刻认识到其在机器学习和深度学习中的重要性，同时提醒我在处理计算高负载、复杂数据时，加速计算、降低内存要求、完成大量推理任务等实际问题常常需要充分利用计算图。