变量作用域实例

变量作用域机制在TensorFlow中主要由两部分组成：

* tf.get\_variable(<name>, <shape>, <initializer>): 通过所给的名字创建或是返回一个变量.
* tf.variable\_scope(<scope\_name>): 通过 tf.get\_variable()为变量名指定命名空间.

方法 tf.get\_variable() 用来获取或创建一个变量，而不是直接调用tf.Variable.它采用的不是像`tf.Variable这样直接获取值来初始化的方法.一个初始化就是一个方法，创建其形状并且为这个形状提供一个张量.这里有一些在TensorFlow中使用的初始化变量：

* tf.constant\_initializer(value) 初始化一切所提供的值,
* tf.random\_uniform\_initializer(a, b)从a到b均匀初始化,
* tf.random\_normal\_initializer(mean, stddev) 用所给平均值和标准差初始化均匀分布.

为了了解tf.get\_variable()怎么解决前面所讨论的问题,让我们在单独的方法里面创建一个卷积来重构一下代码，命名为conv\_relu：

def conv\_relu(input, kernel\_shape, bias\_shape):

# Create variable named "weights".

weights = tf.get\_variable("weights", kernel\_shape,

initializer=tf.random\_normal\_initializer())

# Create variable named "biases".

biases = tf.get\_variable("biases", bias\_shape,

initializer=tf.constant\_intializer(0.0))

conv = tf.nn.conv2d(input, weights,

strides=[1, 1, 1, 1], padding='SAME')

return tf.nn.relu(conv + biases)

这个方法中用了"weights" 和"biases"两个简称.而我们更偏向于用conv1 和 conv2这两个变量的写法,但是不同的变量需要不同的名字.这就是tf.variable\_scope() 变量起作用的地方.他为变量指定了相应的命名空间.

def my\_image\_filter(input\_images):

with tf.variable\_scope("conv1"):

# Variables created here will be named "conv1/weights", "conv1/biases".

relu1 = conv\_relu(input\_images, [5, 5, 32, 32], [32])

with tf.variable\_scope("conv2"):

# Variables created here will be named "conv2/weights", "conv2/biases".

return conv\_relu(relu1, [5, 5, 32, 32], [32])

现在，让我们看看当我们调用 my\_image\_filter() 两次时究竟会发生了什么.

result1 = my\_image\_filter(image1)

result2 = my\_image\_filter(image2)

# Raises ValueError(... conv1/weights already exists ...)

就像你看见的一样，tf.get\_variable()会检测已经存在的变量是否已经共享.如果你想共享他们，你需要像下面使用的一样，通过reuse\_variables()这个方法来指定.

with tf.variable\_scope("image\_filters") as scope:

result1 = my\_image\_filter(image1)

scope.reuse\_variables()

result2 = my\_image\_filter(image2)

用这种方式来共享变量是非常好的，轻量级而且安全.

变量作用域是怎么工作的？

理解 tf.get\_variable()

为了理解变量作用域，首先完全理解tf.get\_variable()是怎么工作的是很有必要的. 通常我们就是这样调用tf.get\_variable 的.

v = tf.get\_variable(name, shape, dtype, initializer)

此调用做了有关作用域的两件事中的其中之一，方法调入.总的有两种情况.

* 情况1:当tf.get\_variable\_scope().reuse == False时，作用域就是为创建新变量所设置的.

这种情况下，v将通过tf.Variable所提供的形状和数据类型来重新创建.创建变量的全称将会由当前变量作用域名+所提供的名字所组成,并且还会检查来确保没有任何变量使用这个全称.如果这个全称已经有一个变量使用了，那么方法将会抛出ValueError错误.如果一个变量被创建,他将会用initializer(shape)进行初始化.比如：

with tf.variable\_scope("foo"):

v = tf.get\_variable("v", [1])

assert v.name == "foo/v:0"

* 情况1：当tf.get\_variable\_scope().reuse == True时，作用域是为重用变量所设置

这种情况下，调用就会搜索一个已经存在的变量，他的全称和当前变量的作用域名+所提供的名字是否相等.如果不存在相应的变量，就会抛出ValueError 错误.如果变量找到了，就返回这个变量.如下：

with tf.variable\_scope("foo"):

v = tf.get\_variable("v", [1])

with tf.variable\_scope("foo", reuse=True):

v1 = tf.get\_variable("v", [1])

assert v1 == v

tf.variable\_scope() 基础

知道tf.get\_variable()是怎么工作的,使得理解变量作用域变得很容易.变量作用域的主方法带有一个名称，它将会作为前缀用于变量名,并且带有一个重用标签来区分以上的两种情况.嵌套的作用域附加名字所用的规则和文件目录的规则很类似：

with tf.variable\_scope("foo"):

with tf.variable\_scope("bar"):

v = tf.get\_variable("v", [1])

assert v.name == "foo/bar/v:0"

当前变量作用域可以用tf.get\_variable\_scope()进行检索并且reuse 标签可以通过调用tf.get\_variable\_scope().reuse\_variables()设置为True .

with tf.variable\_scope("foo"):

v = tf.get\_variable("v", [1])

tf.get\_variable\_scope().reuse\_variables()

v1 = tf.get\_variable("v", [1])

assert v1 == v

注意你*不能*设置reuse标签为False.其中的原因就是允许改写创建模块的方法.想一下你前面写得方法my\_image\_filter(inputs).有人在变量作用域内调用reuse=True 是希望所有内部变量都被重用.如果允许在方法体内强制执行reuse=False，将会打破内部结构并且用这种方法使得很难再共享参数.

即使你不能直接设置 reuse 为 False ,但是你可以输入一个重用变量作用域,然后就释放掉,就成为非重用的变量.当打开一个变量作用域时,使用reuse=True 作为参数是可以的.但也要注意，同一个原因，reuse 参数是不可继承.所以当你打开一个重用变量作用域，那么所有的子作用域也将会被重用.

with tf.variable\_scope("root"):

# At start, the scope is not reusing.

assert tf.get\_variable\_scope().reuse == False

with tf.variable\_scope("foo"):

# Opened a sub-scope, still not reusing.

assert tf.get\_variable\_scope().reuse == False

with tf.variable\_scope("foo", reuse=True):

# Explicitly opened a reusing scope.

assert tf.get\_variable\_scope().reuse == True

with tf.variable\_scope("bar"):

# Now sub-scope inherits the reuse flag.

assert tf.get\_variable\_scope().reuse == True

# Exited the reusing scope, back to a non-reusing one.

assert tf.get\_variable\_scope().reuse == False

获取变量作用域

在上面的所有例子中，我们共享参数只因为他们的名字是一致的，那是因为我们开启一个变量作用域重用时刚好用了同一个字符串.在更复杂的情况，他可以通过变量作用域对象来使用，而不是通过依赖于右边的名字来使用.为此,变量作用域可以被获取并使用，而不是仅作为当开启一个新的变量作用域的名字.

with tf.variable\_scope("foo") as foo\_scope:

v = tf.get\_variable("v", [1])

with tf.variable\_scope(foo\_scope)

w = tf.get\_variable("w", [1])

with tf.variable\_scope(foo\_scope, reuse=True)

v1 = tf.get\_variable("v", [1])

w1 = tf.get\_variable("w", [1])

assert v1 == v

assert w1 == w

当开启一个变量作用域，使用一个预先已经存在的作用域时，我们会跳过当前变量作用域的前缀而直接成为一个完全不同的作用域.这就是我们做得完全独立的地方.

with tf.variable\_scope("foo") as foo\_scope:

assert foo\_scope.name == "foo"

with tf.variable\_scope("bar")

with tf.variable\_scope("baz") as other\_scope:

assert other\_scope.name == "bar/baz"

with tf.variable\_scope(foo\_scope) as foo\_scope2:

assert foo\_scope2.name == "foo" # Not changed.

变量作用域中的初始化器

使用tf.get\_variable()允许你重写方法来创建或者重用变量,并且可以被外部透明调用.但是如果我们想改变创建变量的初始化器那要怎么做呢？是否我们需要为所有的创建变量方法传递一个额外的参数呢？那在大多数情况下，当我们想在一个地方并且为所有的方法的所有的变量设置一个默认初始化器，那又改怎么做呢？为了解决这些问题，变量作用域可以携带一个默认的初始化器.他可以被子作用域继承并传递给tf.get\_variable() 调用.但是如果其他初始化器被明确地指定,那么他将会被重写.

with tf.variable\_scope("foo", initializer=tf.constant\_initializer(0.4)):

v = tf.get\_variable("v", [1])

assert v.eval() == 0.4 # Default initializer as set above.

w = tf.get\_variable("w", [1], initializer=tf.constant\_initializer(0.3)):

assert w.eval() == 0.3 # Specific initializer overrides the default.

with tf.variable\_scope("bar"):

v = tf.get\_variable("v", [1])

assert v.eval() == 0.4 # Inherited default initializer.

with tf.variable\_scope("baz", initializer=tf.constant\_initializer(0.2)):

v = tf.get\_variable("v", [1])

assert v.eval() == 0.2 # Changed default initializer.

在tf.variable\_scope()中ops的名称

我们讨论 tf.variable\_scope 怎么处理变量的名字.但是又是如何在作用域中影响到 其他ops的名字的呢？ops在一个变量作用域的内部创建，那么他应该是共享他的名字，这是很自然的想法.出于这样的原因，当我们用with tf.variable\_scope("name")时，这就间接地开启了一个tf.name\_scope("name").比如：

with tf.variable\_scope("foo"):

x = 1.0 + tf.get\_variable("v", [1])

assert x.op.name == "foo/add"

名称作用域可以被开启并添加到一个变量作用域中,然后他们只会影响到ops的名称,而不会影响到变量.

with tf.variable\_scope("foo"):

with tf.name\_scope("bar"):

v = tf.get\_variable("v", [1])

x = 1.0 + v

assert v.name == "foo/v:0"

assert x.op.name == "foo/bar/add"

当用一个引用对象而不是一个字符串去开启一个变量作用域时，我们就不会为ops改变当前的名称作用域.