TF 变量共享 – TF Variables Sharing

原文：https://www.tensorflow.org/versions/r0.9/how_tos/variable_scope/index.html

你可以像我们在一开始那样，用 tf.Variable 来直接的创建、初始化、保存一集载入单个单个的变量。但当你在建立一个复杂的模型时，你时常需要与其它模型或者当前模型的其它成分共享大量的变量，且你可能会想把这些变量在一个地方同时地初始化。这个教程将向你演示如何使用tf.variable_scope()与tf.get_variable()来完成上述的这些需求。

遇到的问题

想象一下当你创建一个简单的模型来分类图像，就比如我们在CNN教程中那样的模型，不同的是这个简单模型将汁含有两个卷积层（简单起见在这个例子中只使用两个卷积层）。如果你像我们在之前的教程中那样只用tf.Variable来构建这个模型，那么你这个模型的代码可能会长这样：

```
def my_image_filter(input_images):

    conv1_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv1_weights")
    conv1_biases = tf.Variable(tf.zeros([32]), name="conv1_biases")
    conv1 = tf.nn.conv2d(input_images, conv1_weights, strides=[1, 1, 1, 1], padding='SAME')
    relu1 = tf.nn.relu(conv1 + conv1_biases)

    conv2_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv2_weights")
    conv2_biases = tf.Variable(tf.zeros([32]), name="conv2_biases")
    conv2 = tf.nn.conv2d(relu1, conv2_weights,  strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv2 + conv2_biases)
```

而这只是一个十分简单的模型，正如你所想的那样，我们的模型将很快变的比这个模型复杂许多。而且仅仅是在这么一个简单的模型中，我们已经有了conv1_weights, conv1_biases, conv2_weights与conv2_biases四个变量了。

当你想重用这个模型的时候，那么问题来了：假设你想把你的这个模型应用在两个不同的图片上，比如image1与image2，你想以一模一样的参数来处理这两张图片，你能想到的是调用两次my_image_filter()来处理这两张图片，就像下面的这段代码展示的那样，但这存在一个问题，你的代码创建了两套不同的参数，这并不是你想要的。

```
# First call creates one set of variables.
result1 = my_image_filter(image1)
# Another set is created in the second call.
result2 = my_image_filter(image2)
```

那么如何共享你的参数呢？正如下面这段代码展示的那样，一个比较常见的方法是用一个专门的容器去存储你的参数，然后在使用my_image_filter时载入该容器中的参数，这样两个图片分类器使用的参数就一样了呢。

```
variables_dict = {
    "conv1_weights" : tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv1_weights")
    "conv1_biases"  : tf.Variable(tf.zeros([32]), name="conv1_biases")
    ... etc. ...
}

def my_image_filter(input_images, variables_dict):

    conv1 = tf.nn.conv2d(input_images, variables_dict["conv1_weights"],
                         strides=[1, 1, 1, 1], padding='SAME')
    relu1 = tf.nn.relu(conv1 + variables_dict["conv1_biases"])

    conv2 = tf.nn.conv2d(relu1, variables_dict["conv2_weights"],
                         strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv2 + variables_dict["conv2_biases"])

# The 2 calls to my_image_filter() now use the same variables
result1 = my_image_filter(image1, variables_dict)
result2 = my_image_filter(image2, variables_dict)
```

这样子做虽然很方便，但是破坏了整个模型的封装性:
1. 这段代码的计算图(Graph)必须记录变量的名称，类型与形状，才可以载入变量；
2. 当代码改变时，比如创建更多、更少的、或者其它别的变量（，会不便捷）

那么如何解决这个问题呢？一种办法是，创建一些类来创建模型，并且由这些类管理自己的模型所需要的变量。类似于使用类来管理模型的变量，还有一个更轻量级的解决办法，使用TF提供的变量域机制在创建计算图时十分便捷地共享已命名的这些变量。

变量域实例

TF中的变量域机制主要依靠两个函数：
1. tf.get_variable(<name>, <shape>, <initializer>): 创建或者返回指定名字的变量。
2. tf.variable_scope(<scope_name>): 管理与get_variable()交互的命名空间。

第一个函数用于获取或者创建一个变量，而非直接调用tf.Variable来创建一个变量对象。它使用一个初始化

器而非像tf.Variable那样直接传参初始化。这里所说的初始化器，事实上类似一种映射关系，它接受一个变量对象指定的大小形状，然后返回一个该指定大小形状的张量。下面例举一些TF中提供的初始化器：
1. tf.constant_initializer(value) 使用指定值对所有对象进行初始化；
2. tf.random_uniform_initializer(a, b) 随机均匀地以区间[a, b]中的值初始化对象；
3. tf.random_normal_initializer(mean, stddev) 随机正态地初始化对象。

为了理解tf.get_variable函数是如何解决我们所说的问题的，我们用这个函数来重构一下上一段代码，使用一个conv_relu函数来创建一个完整的卷积层，看完这段代码你将初步了解如何使用这个函数：

```
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_initializer(0.0))
    conv = tf.nn.conv2d(input, weights, strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)
```

在这段新的函数代码中，我们使用了简洁的名称weights与biases来命名权重与偏移量，而非如之前的conv1weights或conv2biases，所以即使我们将会有两个卷积层，但他们权重与偏移量的名称都叫weights与biases。你可能会担心遇到冲突，所以这时我们将使用yf.variable_scope函数来限定区分每个卷积层的变量名称，即将不同的权重与偏移量分配至对应的不同命名空间来解决重名。

```
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])
```

现在让我们再来看，当我们调用两次my_image_filter()时发生了什么。

```
result1 = my_image_filter(image1)
result2 = my_image_filter(image2)
# Raises ValueError(... conv1/weights already exists ...)
```

你会发现，tf.get_variable函数会先检查已经存在的但却并未被设置为共享的变量，并报错，上述例子中将

在result2的计算阶段报错。如果你想要共享这些已存在的变量，你需要使用reuse_variables函数来专门的设置他们使之可被共享：

```
with tf.variable_scope("image_filters") as scope:
    result1 = my_image_filter(image1)
    scope.reuse_variables()
    result2 = my_image_filter(image2)
```

这是一个非常好的办法来共享我们的这些变量，轻量级且安全。

变量域是怎么工作的？

理解tf.get_variable函数

为了理解变量域，我们必须线全面地理解清楚tf.get_variable函数是怎么工作的。下面的示例代码展示了tf.get_variable函数的一般调用方法:

```
v = tf.get_variable(name, shape, dtype, initializer)
```

这个函数的调用两种不同的模式，函数调用时需对被调用变量域选择一种模式。以下是两种模式的说明：
模式1: 当前变量域是用来创建新的变量的，我们可以以以下的语句来判明这一模式。

```
tf.get_variable_scope().reuse == False.
```

在这一模式中，v将会根据其指明的大小与数据类型被创建为一个全新的tf.Variable。该新创建的变量的全称将会被设置为当前变量域名加上该变量名，同时函数将会进行一次重名检查以保证该变量的全称目前是唯一的。假如该变量的全称已经被已存在的另一个变量占用了，则函数会报出ValueError错误。如果一个变量被正常创建了，那么它将被初始化为一个initializer(shape)的值。下面举例说明：

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"
```

模式2: 当前变量域将要重用已存在的变量，我们可以以以下语句判明这一模式。

```
tf.get_variable_scope().reuse == True.
```

在这一模式中，get_variable函数的调用会进行一次搜索，以搜索与被查询变量有相同全称的已有变量。假如没有找到与被查询变量具有相同全称的已有变量，将出现一个ValueError错误。如果正常搜索到了该变量，则函数将返回这个变量。下面举例说明：

```
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函数的基本知识

现在你已经了解了tf.get_variable函数是如何工作的了，这将有助于你理解变量域。变量域的基本用途，一是设置一个域名作为变量的前缀来区分变量的所在域，二是设置一个重用标记位来指明变量域目前应处于上述两种模式中的哪一种。另外，变量域采用一种类似于目录命名的规则来处理嵌套式的变量域名叠加，下面举例说明：

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

我们使用tf.get_variable_scope函数来获取当前变量域的名称，并且我们可以使用tf.get_variable_scope().reuse_variables()函数来设置当前变量域的重用标记位为真。下面举例说明：

```
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
```

需要注意的一点是，你无法设置当前变量域的重用标记位为假。这其中隐含的原因与一些创建模型需用到的函数有关。想象一下你像刚才那样写了一个my_image_filter(inputs)函数。有人在一个重用标记位为真的变量域中调用了该函数，那么他一定会认为在他的这个变量域内所有的变量都是可重用的。如果存在一个可在内部强行重新设置重用标记位为假的方法，将会破坏上面所说的这种既定约定，并使的参数的共享变得困难。

虽然你不能显性地设置作用域的重用标记位为假，但是当你想要不共享当前作用域内的变量时，你可以在程序中退出重用标记位为真的变量域并重新打开一个重用标记位为假的该变量域。需要注意的是，与上一段中描述的原因一样，新进入变量域内所有的变量与子变量域内的变量的重用标记位都被继承性地设置为了假。再一次说明，当你打开一个变量域的重用标记位为真时，所有子变量域的重用标记位也均设置为真了

```
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
```

获取变量域

在上述所举的所有例子中，我们共享参数的依据仅仅是变量拥有相同的全称，我们都是以一个名字来匹配变量域的。在一些更加复杂的例子中，直接通过传递变量域实例来打开变量域可能会更有用一些，而不是像刚才那样每次打开变量域的时候都用一个名称字符串来查询着打开它。出于这一目的，我们在打开一个变量域时可以获取一个变量域对象，以替换使用变量域名字符串。下面举例说明：

```
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
```

当我们我们打开一个以前已经创建好的变量域，我们将跳出当前的变量域，并使用一个完全不同的变量域名称前缀。但这也决定域我们打开变量域的位置，比如下面这两种情况：

```
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.
```

变量域中的初始化器

tf.get_variable()函数让我们的函数可以创建、重用一些变量，并使的这些变量可以在当前函数外的其它地方被使用。但假如我们想修改这些已创建变量的初始化器怎么办？我们是不是需要在创建变量的时候再传入一个额外的参数来指明这个初始化器？在一种最常见的情况中，当我们想为所有的变量一次性设置一个默认初始化器，是应该在所有函数之外做这个设置吗？为了解决这一情况，变量域被创造为可以携带一个初始化器。这个初始化器将被该变量域的子域继承并传递给所有在该域中的tf.get_variable()函数。但需要注意的一点是，该默认初始化器可以被一个显式指明的初始化器覆盖。请看下面这个例子：

```
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.
```

tf.variable_scope()函数中操作符的名称

在上面，我们讨论了tf.variable_scope函数hsi如何管理变量名称的。但是它会影响变量域中操作符的名称吗？直观上来说，在变量域中创建的操作符应该也要共享变量域的名称。所以出于这一点的考虑，当我们用with语句打开了一个名为name的变量域，这将隐式地打开一个名为name的名称域。下面举例说明：

```
with tf.variable_scope("foo"):
   x = 1.0 + tf.get_variable("v", [1])
assert x.op.name == "foo/add"
```

另外，在变量域中可以额外地打开更多的名称域，且他们将只作用域操作符，而不会影响变量。

```
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"
```

当我们用变量域对象而非变量域名城字符串打开一个变量域的时候，当前名称域将不会被修改。

使用的实例

下面我们给出一些使用了变量域的实例的文件。这在RNN与序列映射模型中尤其常用。

文件	内容
models/image/cifar10.py	识别图片中对象的模型 。
models/rnn/rnn_cell.py	RNN模型单元。
models/rnn/seq2seq.py	序列映射模型。