Sharing Variables

You can create, initialize, save and load single variables in the way described in the Variables HowTo (https://www.tensorflow.org/programmers_guide/variables). But when building complex models you often need to share large sets of variables and you might want to initialize all of them in one place. This tutorial shows how this can be done using `tf.variable_scope()` and `tf.get_variable()`.

The Problem

Imagine you create a simple model for image filters, similar to our Convolutional Neural Networks Tutorial (https://www.tensorflow.org/tutorials/deep_cnn) model but with only 2 convolutions (for simplicity of this example). If you use just `tf.Variable`, as explained in Variables HowTo (https://www.tensorflow.org/programmers_guide/variables), your model might look like this.

```
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)
```

As you can easily imagine, models quickly get much more complicated than this one, and even here we already have 4 different variables: `conv1_weights`, `conv1_biases`, `conv2_weights`, and `conv2_biases`.

The problem arises when you want to reuse this model. Assume you want to apply your image filter to 2 different images, `image1` and `image2`. You want both images processed by the same filter with the same parameters. You can call `my_image_filter()` twice, but this will create two sets of variables, 4 variables in each one, for a total of 8 variables.

```
# First call creates one set of 4 variables.
result1 = my_image_filter(image1)
# Another set of 4 variables is created in the second call.
result2 = my_image_filter(image2)
```

A common way to share variables is to create them in a separate piece of code and pass them to functions that use them. For example by using a dictionary:

```
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"])

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

While convenient, creating variables like above, outside of the code, breaks encapsulation:

- The code that builds the graph must document the names, types, and shapes of variables to create.
- When the code changes, the callers may have to create more, or less, or different variables.

One way to address the problem is to use classes to create a model, where the classes take care of managing the variables they need. For a lighter solution, not involving classes, TensorFlow provides a *Variable Scope* mechanism that allows to easily share named variables while constructing a graph.

Variable Scope Example

Variable Scope mechanism in TensorFlow consists of two main functions:

- `tf.get_variable(<name>, <shape>, <initializer>)`: Creates or returns a variable with a given name.
- `tf.variable_scope(<scope_name>)`: Manages namespaces for names passed to `tf.get_variable()`.

The function `tf.get_variable()` is used to get or create a variable instead of a direct call to `tf.Variable`. It uses an *initializer* instead of passing the value directly, as in `tf.Variable`. An initializer is a function that takes the shape and provides a tensor with that shape. Here are some initializers available in TensorFlow:

- `tf.constant_initializer(value)` initializes everything to the provided value,
- `tf.random_uniform_initializer(a, b)` initializes uniformly from [a, b],
- `tf.random_normal_initializer(mean, stddev)` initializes from the normal distribution with the given mean and standard deviation.

To see how `tf.get_variable()` solves the problem discussed before, let's refactor the code that created one convolution into a separate function, named `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_initializer(0.0))
    conv = tf.nn.conv2d(input, weights,
        strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)
```

This function uses short names `"weights"` and `"biases"`. We'd like to use it for both `conv1` and `conv2`, but the variables need to have different names. This is where `tf.variable_scope()` comes into play: it pushes a namespace for variables.

```
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])
```

Now, let's see what happens when we call `my_image_filter()` twice.

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

As you can see, `tf.get_variable()` checks that already existing variables are not shared by accident. If you want to share them, you need to specify it by setting `reuse_variables()` as follows.

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

This is a good way to share variables, lightweight and safe.

How Does Variable Scope Work?

Understanding `tf.get_variable()`

To understand variable scope it is necessary to first fully understand how `tf.get_variable()` works. Here is how `tf.get_variable` is usually called.

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

This call does one of two things depending on the scope it is called in. Here are the two options.

- Case 1: the scope is set for creating new variables, as evidenced by `tf.get_variable_scope().reuse == False`.

In this case, `v` will be a newly created `tf.Variable` with the provided shape and data type. The full name of the created variable will be set to the current variable scope name + the provided `name` and a check will be performed to ensure that no variable with this full name exists yet. If a variable with this full name already exists, the function will raise a `ValueError`. If a new variable is created, it will be initialized to the value `initializer(shape)`. For example:

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

- Case 2: the scope is set for reusing variables, as evidenced by `tf.get_variable_scope().reuse == True`.

In this case, the call will search for an already existing variable with name equal to the current variable scope name + the provided `name`. If no such variable exists, a `ValueError` will be raised. If the variable is found, it will be returned. For example:

```
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 is v
```

Basics of `tf.variable_scope()`

Knowing how `tf.get_variable()` works makes it easy to understand variable scope. The primary function of variable scope is to carry a name that will be used as prefix for variable names and a reuse-flag to distinguish the two cases described above. Nesting variable scopes appends their names in a way analogous to how directories work:

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

The current variable scope can be retrieved using `tf.get_variable_scope()` and the `reuse` flag of the current variable scope can be set to `True` by calling `tf.get_variable_scope().reuse_variables()`:

```
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 is v
```

Note that you *cannot* set the `reuse` flag to `False`. The reason behind this is to allow to compose functions that create models. Imagine you write a function `my_image_filter(inputs)` as before. Someone calling the function in a variable scope with `reuse=True` would expect all inner variables to be reused as well. Allowing to force `reuse=False` inside the function would break this contract and make it hard to share parameters in this way.

Even though you cannot set `reuse` to `False` explicitly, you can enter a reusing variable scope and then exit it, going back to a non-reusing one. This can be done using a `reuse=True` parameter when opening a variable scope. Note also that, for the same reason as above, the `reuse` parameter is inherited. So when you open a reusing variable scope, all sub-scopes will be reusing too.

```
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
```

Capturing variable scope

In all examples presented above, we shared parameters only because their names agreed, that is, because we opened a reusing variable scope with exactly the same string. In more complex cases, it might be useful to pass a VariableScope object rather than rely on getting the names right. To this end, variable scopes can be captured and used instead of names when opening a new variable scope.

```
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 is v
assert w1 is w
```

When opening a variable scope using a previously existing scope we jump out of the current variable scope prefix to an entirely different one. This is fully independent of where we do it.

```
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.
```

Initializers in variable scope

Using `tf.get_variable()` allows to write functions that create or reuse variables and can be transparently called from outside. But what if we wanted to change the initializer of the created variables? Do we need to pass an extra argument to every function that creates variables? What about the most common case, when we want to set the default initializer for all variables in one place, on top of all functions? To help with these cases, variable scope can carry a default initializer. It is inherited by sub-scopes and passed to each `tf.get_variable()` call. But it will be overridden if another initializer is specified explicitly.

```
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.
```

Names of ops in `tf.variable_scope()`

We discussed how `tf.variable_scope` governs the names of variables. But how does it influence the names of other ops in the scope? It is natural that ops created inside a variable scope should also share that name. For this reason, when we do `with tf.variable_scope("name")`, this implicitly opens a `tf.name_scope("name")`. For example:

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

Name scopes can be opened in addition to a variable scope, and then they will only affect the names of the ops, but not of variables.

```
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"
```

When opening a variable scope using a captured object instead of a string, we do not alter the current name scope for ops.

Examples of Use

Here are pointers to a few files that make use of variable scope. They can all be found in the TensorFlow models repo (https://github.com/tensorflow/models). In particular, variable scope is heavily used for recurrent neural networks and sequence-to-sequence models.

File	What's in it?
`tutorials/image/cifar10/cifar10.py`	Model for detecting objects in images.
`tutorials/rnn/rnn_cell.py`	Cell functions for recurrent neural networks.
`tutorials/rnn/seq2seq.py`	Functions for building sequence-to-sequence models.

上次更新日期：六月 19, 2017