Using name scopes, you can make the graph much clearer. Simply move all the content of the `relu()` function inside a name scope. Figure 9-7 shows the resulting graph. Notice that TensorFlow also gives the name scopes unique names by appending `_1`, `_2`, and so on.

```
def relu(X):
    with tf.name_scope("relu"):
        [...]
```

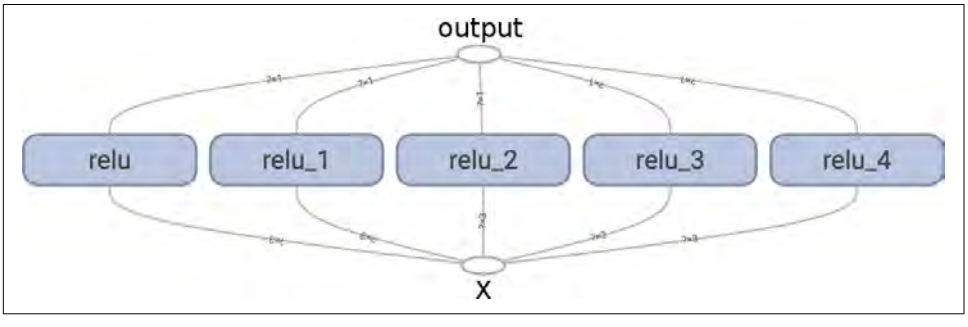

Figure 9-7. A clearer graph using name-scoped units

Sharing Variables

If you want to share a variable between various components of your graph, one simple option is to create it first, then pass it as a parameter to the functions that need it. For example, suppose you want to control the ReLU threshold (currently hardcoded to 0) using a shared `threshold` variable for all ReLUs. You could just create that variable first, and then pass it to the `relu()` function:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

This works fine: now you can control the threshold for all ReLUs using the `threshold` variable. However, if there are many shared parameters such as this one, it will be painful to have to pass them around as parameters all the time. Many people create a Python dictionary containing all the variables in their model, and pass it around to every function. Others create a class for each module (e.g., a `ReLU` class using class variables to handle the shared parameter). Yet another option is to set the shared variable as an attribute of the `relu()` function upon the first call, like so:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow offers another option, which may lead to slightly cleaner and more modular code than the previous solutions.[5] This solution is a bit tricky to understand at first, but since it is used a lot in TensorFlow it is worth going into a bit of detail. The idea is to use the `get_variable()` function to create the shared variable if it does not exist yet, or reuse it if it already exists. The desired behavior (creating or reusing) is controlled by an attribute of the current `variable_scope()`. For example, the following code will create a variable named `"relu/threshold"` (as a scalar, since `shape=()`, and using `0.0` as the initial value):

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

Note that if the variable has already been created by an earlier call to `get_variable()`, this code will raise an exception. This behavior prevents reusing variables by mistake. If you want to reuse a variable, you need to explicitly say so by setting the variable scope's `reuse` attribute to `True` (in which case you don't have to specify the shape or the initializer):

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

This code will fetch the existing `"relu/threshold"` variable, or raise an exception if it does not exist or if it was not created using `get_variable()`. Alternatively, you can set the `reuse` attribute to `True` inside the block by calling the scope's `reuse_variables()` method:

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```

Once `reuse` is set to `True`, it cannot be set back to `False` within the block. Moreover, if you define other variable scopes inside this one, they will automatically inherit `reuse=True`. Lastly, only variables created by `get_variable()` can be reused this way.

5 Creating a `ReLU` class is arguably the cleanest option, but it is rather heavyweight.

Now you have all the pieces you need to make the `relu()` function access the `threshold` variable without having to pass it as a parameter:

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")  # reuse existing variable
        [...]
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):  # create the variable
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
```

This code first defines the `relu()` function, then creates the `relu/threshold` variable (as a scalar that will later be initialized to `0.0`) and builds five ReLUs by calling the `relu()` function. The `relu()` function reuses the `relu/threshold` variable, and creates the other ReLU nodes.

Variables created using `get_variable()` are always named using the name of their `variable_scope` as a prefix (e.g., `"relu/threshold"`), but for all other nodes (including variables created with `tf.Variable()`) the variable scope acts like a new name scope. In particular, if a name scope with an identical name was already created, then a suffix is added to make the name unique. For example, all nodes created in the preceding code (except the `threshold` variable) have a name prefixed with `"relu_1/"` to `"relu_5/"`, as shown in Figure 9-8.

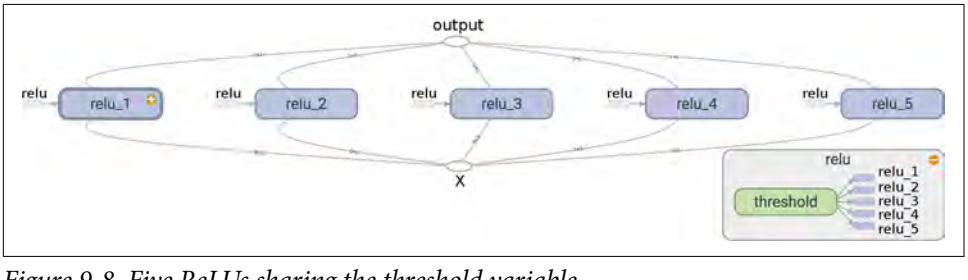

Figure 9-8. Five ReLUs sharing the threshold variable

It is somewhat unfortunate that the `threshold` variable must be defined outside the `relu()` function, where all the rest of the ReLU code resides. To fix this, the following code creates the `threshold` variable within the `relu()` function upon the first call, then reuses it in subsequent calls. Now the `relu()` function does not have to worry about name scopes or variable sharing: it just calls `get_variable()`, which will create

or reuse the `threshold` variable (it does not need to know which is the case). The rest of the code calls `relu()` five times, making sure to set `reuse=False` on the first call, and `reuse=True` for the other calls.

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

The resulting graph is slightly different than before, since the shared variable lives within the first ReLU (see Figure 9-9).

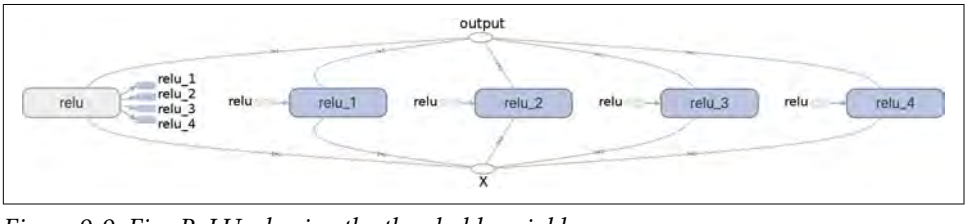

Figure 9-9. Five ReLUs sharing the threshold variable

This concludes this introduction to TensorFlow. We will discuss more advanced topics as we go through the following chapters, in particular many operations related to deep neural networks, convolutional neural networks, and recurrent neural networks as well as how to scale up with TensorFlow using multithreading, queues, multiple GPUs, and multiple servers.

Exercises

1. What are the main benefits of creating a computation graph rather than directly executing the computations? What are the main drawbacks?
2. Is the statement `a_val = a.eval(session=sess)` equivalent to `a_val = sess.run(a)`?
3. Is the statement `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` equivalent to `a_val, b_val = sess.run([a, b])`?
4. Can you run two graphs in the same session?