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).

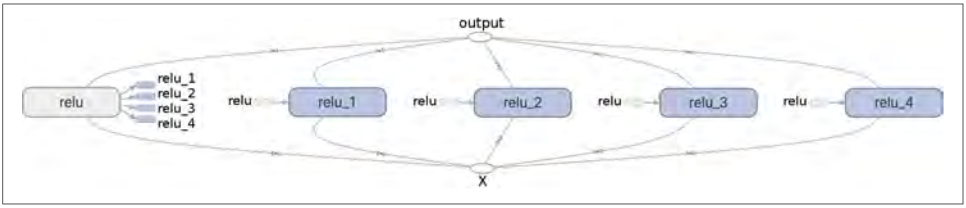


*Figure 9-9. Five ReLUs sharing the threshold variable*

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?

5. If you create a graph `g` containing a variable `w`, then start two threads and open a session in each thread, both using the same graph `g`, will each session have its own copy of the variable `w` or will it be shared?
6. When is a variable initialized? When is it destroyed?
7. What is the difference between a placeholder and a variable?
8. What happens when you run the graph to evaluate an operation that depends on a placeholder but you don't feed its value? What happens if the operation does not depend on the placeholder?
9. When you run a graph, can you feed the output value of any operation, or just the value of placeholders?
10. How can you set a variable to any value you want (during the execution phase)?
11. How many times does reverse-mode autodiff need to traverse the graph in order to compute the gradients of the cost function with regards to 10 variables? What about forward-mode autodiff? And symbolic differentiation?
12. Implement Logistic Regression with Mini-batch Gradient Descent using TensorFlow. Train it and evaluate it on the moons dataset (introduced in Chapter 5). Try adding all the bells and whistles:
    - Define the graph within a `logistic_regression()` function that can be reused easily.
    - Save checkpoints using a `Saver` at regular intervals during training, and save the final model at the end of training.
    - Restore the last checkpoint upon startup if training was interrupted.
    - Define the graph using nice scopes so the graph looks good in TensorBoard.
    - Add summaries to visualize the learning curves in TensorBoard.
    - Try tweaking some hyperparameters such as the learning rate or the mini-batch size and look at the shape of the learning curve.

Solutions to these exercises are available in Appendix A.

# CHAPTER 10
# Introduction to Artificial Neural Networks

Birds inspired us to fly, burdock plants inspired velcro, and nature has inspired many other inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the key idea that inspired *artificial neural networks* (ANNs). However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying "units" rather than "neurons"), lest we restrict our creativity to biologically plausible systems.[1]

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks, such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of *Go* by examining millions of past games and then playing against itself (DeepMind's AlphaGo).

In this chapter, we will introduce artificial neural networks, starting with a quick tour of the very first ANN architectures. Then we will present *Multi-Layer Perceptrons* (MLPs) and implement one using TensorFlow to tackle the MNIST digit classification problem (introduced in Chapter 3).

1 You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.