Introduction to Broadcasting

Broadcasting is a term (introduced by NumPy) for when a tensor system's matrices and vectors of different sizes can be added together. These rules allow for conveniences like adding a vector to every row of a matrix. Broadcasting rules can be quite complex, so we will not dive into a formal discussion of the rules. It's often easier to experiment and see how the broadcasting works (Example 2-18).

Example 2-18. Examples of broadcasting

```
>>> a = tf.ones((2, 2))
>>> a.eval()
array([[ 1.,  1.],
       [ 1.,  1.]], dtype=float32)
>>> b = tf.range(0, 2, 1, dtype=tf.float32)
>>> b.eval()
array([ 0.,  1.], dtype=float32)
>>> c = a + b
>>> c.eval()
array([[ 1.,  2.],
       [ 1.,  2.]], dtype=float32)
```

Notice that the vector `b` is added to every row of matrix `a`. Notice another subtlety; we explicitly set the `dtype` for `b`. If the `dtype` isn't set, TensorFlow will report a type error. Let's see what would have happened if we hadn't set the `dtype` (Example 2-19).

Example 2-19. TensorFlow doesn't perform implicit type casting

```
>>> b = tf.range(0, 2, 1)
>>> b.eval()
array([0, 1], dtype=int32)
>>> c = a + b
ValueError: Tensor conversion requested dtype float32 for Tensor with dtype int32:
'Tensor("range_2:0", shape=(2,), dtype=int32)
```

Unlike languages like C, TensorFlow doesn't perform implicit type casting under the hood. It's often necessary to perform explicit type casts when doing arithmetic operations.

Imperative and Declarative Programming

Most situations in computer science involve imperative programming. Consider a simple Python program (Example 2-20).

Example 2-20. Python program imperatively performing an addition

```
>>> a = 3
>>> b = 4
>>> c = a + b
>>> c
7
```

This program, when translated into machine code, instructs the machine to perform a primitive addition operation on two registers, one containing 3, and the other containing 4. The result is then 7. This style of programming is called *imperative* since the program tells the computer explicitly which actions to perform.

An alternative style of programming is *declarative*. In a declarative system, a computer program is a high-level description of the computation that is to be performed. It does not instruct the computer exactly how to perform the computation. Example 2-21 is the TensorFlow equivalent of Example 2-20.

Example 2-21. TensorFlow program declaratively performing an addition

```
>>> a = tf.constant(3)
>>> b = tf.constant(4)
>>> c = a + b
>>> c
<tf.Tensor 'add_1:0' shape=() dtype=int32>
>>> c.eval()
7
```

Note that the value of `c` isn't `7`! Rather, it's a symbolic tensor. This code specifies the computation of adding two values together to create a new tensor. The actual computation isn't executed until we call `c.eval()`. In the sections before, we have been using the `eval()` method to simulate imperative style in TensorFlow since it can be challenging to understand declarative programming at first.

However, declarative programming is by no means an unknown concept to software engineering. Relational databases and SQL provide an example of a widely used declarative programming system. Commands like SELECT and JOIN may be implemented in an arbitrary fashion under the hood so long as their basic semantics are preserved. TensorFlow code is best thought of as analogous to a SQL program; the TensorFlow code specifies a computation to be performed, with details left up to TensorFlow. The TensorFlow developers exploit this lack of detail under the hood to tailor the execution style to the underlying hardware, be it CPU, GPU, or mobile device.

It's important to note that the grand weakness of declarative programming is that the abstraction is quite leaky. For example, without detailed understanding of the underlying implementation of the relational database, long SQL programs can become unbearably inefficient. Similarly, large TensorFlow programs implemented without

understanding of the underlying learning algorithms are unlikely to work well. In the rest of this section, we will start paring back the abstraction, a process we will continue throughout the rest of the book.

TensorFlow Eager

The TensorFlow team recently added a new experimental module, TensorFlow Eager, that enables users to run TensorFlow calculations imperatively. In time, this module will likely become the preferred entry mode for new programmers learning TensorFlow. However, at the timing of writing, this module is still very new with many rough edges. As a result, we won't teach you about Eager mode, but encourage you to check it out for yourself.

It's important to emphasize that much of TensorFlow will remain declarative even after Eager matures, so it's worth learning declarative TensorFlow regardless.

TensorFlow Graphs

Any computation in TensorFlow is represented as an instance of a `tf.Graph` object. Such a graph consists of a set of instances of `tf.Tensor` objects and `tf.Operation` objects. We have covered `tf.Tensor` in some detail, but what are `tf.Operation` objects? You have already seen them over the course of this chapter. A call to an operation like `tf.matmul` creates a `tf.Operation` instance to mark the need to perform the matrix multiplication operation.

When a `tf.Graph` is not explicitly specified, TensorFlow adds tensors and operations to a hidden global `tf.Graph` instance. This instance can be fetched by `tf.get_default_graph()` (Example 2-22).

Example 2-22. Getting the default TensorFlow graph

```
>>> tf.get_default_graph()
<tensorflow.python.framework.ops.Graph>
```

It is possible to specify that TensorFlow operations should be performed in graphs other than the default. We will demonstrate examples of this in future chapters.

TensorFlow Sessions

In TensorFlow, a `tf.Session()` object stores the context under which a computation is performed. At the beginning of this chapter, we used `tf.InteractiveSession()` to set up an environment for all TensorFlow computations. This call created a hidden global context for all computations performed. We then used `tf.Tensor.eval()` to