Appendix D. TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF functions (see Chapter 12).

TF Functions and Concrete Functions

TF functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<ConcreteFunction tf_cube(x) at 0x7F84411F4250>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Figure D-1 shows the `tf_cube()` TF function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (`FuncGraph`) and its own *function definition* (`FunctionDef`). A function definition points to the parts of the graph that correspond to the function's inputs and outputs. In each `FuncGraph`, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like `x`), while the edges (the solid arrows between the operations) represent the tensors that will flow through the graph. The concrete function on the left is specialized for `x=2`, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for `x` will output 5.0, then the power operation will compute `5.0 ** 3`, so the output will be 125.0.

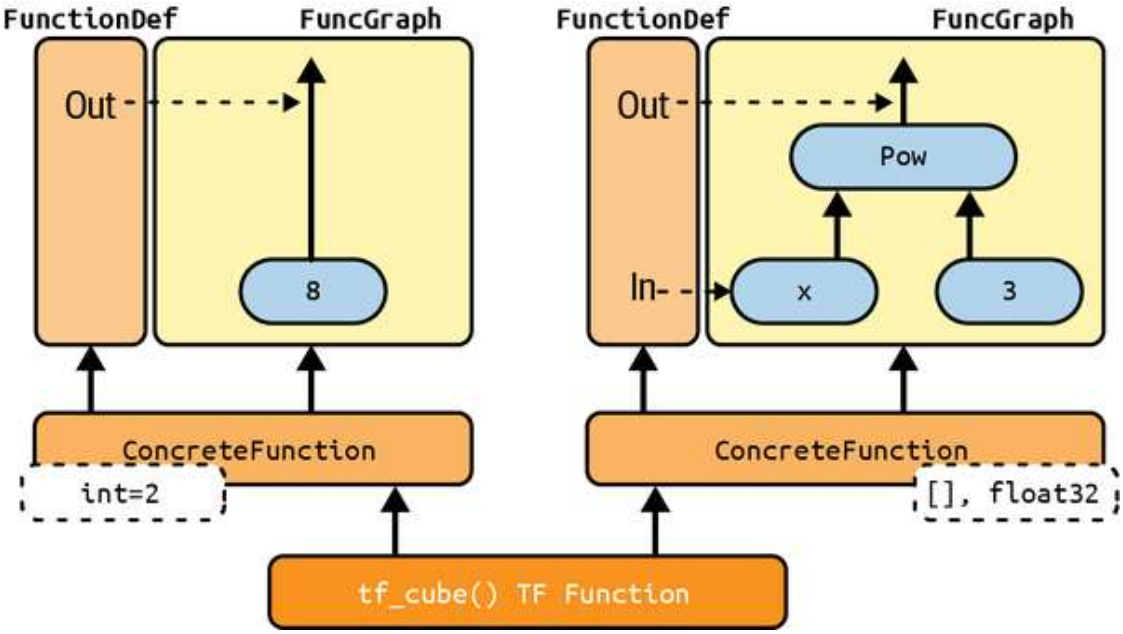

Figure D-1. The `tf_cube()` TF function, with its `ConcreteFunction`s and their `FuncGraph`s

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder `x` and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function's computation graph using the `graph` attribute, and get the list of its operations by calling the graph's `get_operations()` method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x7f84411f4790>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument `x` (it is called a *placeholder*), the second "operation" represents the constant `3`, the third operation represents the power operation (`**`), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the power operation[1]). Each operation has a list of input and output tensors that you can easily access using the operation's `inputs` and `outputs` attributes.

For example, let's get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in Figure D-2.

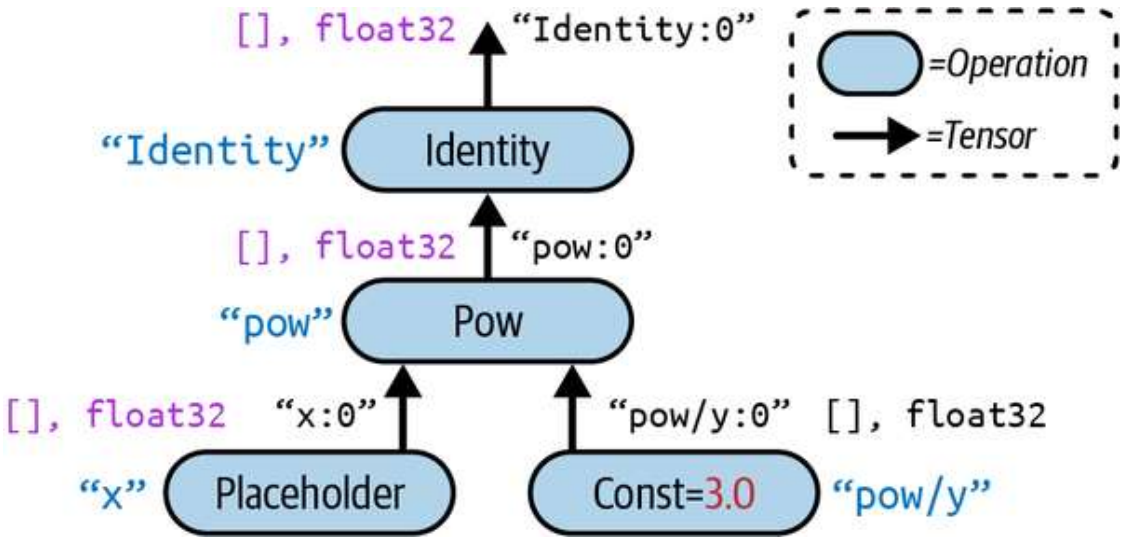

Figure D-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., `"pow"`), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., `"pow_1"`, `"pow_2"`, etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus `:0` if it is the operation's first output, or `:1` if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph's `get_operation_by_name()` or `get_tensor_by_name()` methods:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
```

```
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

The concrete function also contains the function definition (represented as a protocol buffer[2]), which includes the function's signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```
>>> concrete_function.function_def.signature
name: "__inference_tf_cube_3515903"
input_arg {
  name: "x"
  type: DT_FLOAT
}
output_arg {
  name: "identity"
  type: DT_FLOAT
}
```

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print(f"x = {x}")
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

The `result` looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name (`"x:0"`). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2)  # new Python value: trace!
x = 2
>>> result = tf_cube(3)  # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]]))  # new shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]]))  # new shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]]))  # same shape: no trace
```

WARNING

If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF function is called with a new input signature). It's best to assume that the function may be traced (or not) any time the TF function is called.

In some cases, you may want to restrict a TF function to a specific input signature. For example, suppose you know that you will only ever call a TF function with batches of 28 × 28–pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a dif-

ferent concrete function for each batch size, or count on it to figure out on its own when to use `None`. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2]  # drop half the rows and columns
```

This TF function will accept any float32 tensor of shape [*, 28, 28], and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1)  # works fine, traces the function
preprocessed_images = shrink(img_batch_2)  # works fine, same concrete function
```

However, if you try to call this TF function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3)  # ValueError! Incompatible inputs
```

Using AutoGraph to Capture Control Flow

If your function contains a simple `for` loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: shape=(), dtype=int32, numpy=15>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=AddV2>, [...],
 <tf.Operation 'add_1' type=AddV2>, [...],
 <tf.Operation 'add_2' type=AddV2>, [...],
 [...]
 <tf.Operation 'add_9' type=AddV2>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this `for` loop as a "static" loop that gets unrolled when the graph is created.

If you want the graph to contain a "dynamic" loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.while_loop()` operation, but it is not very intuitive (see the "Using AutoGraph to Capture Control Flow" section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow's *AutoGraph* feature, discussed in Chapter 12. AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`). So if it is on, why didn't it capture the `for` loop in the `add_10()` function? It only captures `for` loops that iterate over tensors of `tf.data.Dataset` objects, so you should use `tf.range()`, not `range()`. This is to give you the choice:

- If you use `range()`, the `for` loop will be static, meaning it will only be executed when the function is traced. The loop will be "unrolled" into a set of operations for each iteration, as we saw.
- If you use `tf.range()`, the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let's look at the graph that gets generated if we just replace `range()` with `tf.range()` in the `add_10()` function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'while' type=StatelessWhile>, [...]]
```

As you can see, the graph now contains a `While` loop operation, as if we had called the `tf.while_loop()` function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF function, it gets passed by reference, so the function may modify it. For example:

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter)  # counter is now equal to 1
increment(counter)  # counter is now equal to 2
```

If you peek at the function definition, the first argument is marked as a resource:

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
```

```
  name: "counter"
  type: DT_RESOURCE
```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

The TF function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is `@tf.function` works fine with methods too:

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)
```

WARNING

Do not use `=`, `+=`, `-=`, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, Keras. Let's see how to use TF functions with Keras.

Using TF Functions with Keras (or Not)

By default, any custom function, layer, or model you use with Keras will automatically be converted to a TF function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if your custom code cannot be turned into a TF function, or if you just want to debug your code (which is much easier in eager mode). To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyDense(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

Now you know how TF functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF functions with Keras.

1 You can safely ignore it—it is only here for technical reasons, to ensure that TF functions don't leak internal structures.

2 A popular binary format discussed in Chapter 13.