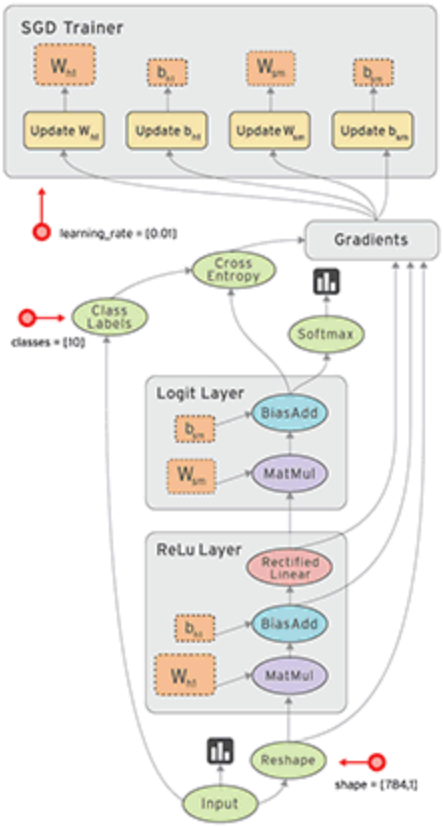
Graphs and Sessions

TensorFlow uses a **dataflow graph** to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow **session** to run parts of the graph across a set of local and remote devices.

This guide will be most useful if you intend to use the low-level programming model directly. Higher-level APIs such as `tf.estimator.Estimator` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) and Keras hide the details of graphs and sessions from the end user, but this guide may also be useful if you want to understand how these APIs are implemented.

Why dataflow graphs?



Dataflow (https://en.wikipedia.org/wiki/Dataflow_programming) is a common programming model for parallel computing. In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. For example, in a TensorFlow graph, the `tf.matmul` (https://www.tensorflow.org/api_docs/python/tf/linalg/matmul) operation would correspond to a single node with two incoming edges (the matrices to be multiplied) and one outgoing edge (the result of the multiplication).

Dataflow has several advantages that TensorFlow leverages when executing your programs:

- **Parallelism.** By using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel.
- **Distributed execution.** By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to

different machines. TensorFlow inserts the necessary communication and coordination between devices.

- **Compilation.** TensorFlow's XLA compiler (https://www.tensorflow.org/performance/xla/index) can use the information in your dataflow graph to generate faster code, for example, by fusing together adjacent operations.
- **Portability.** The dataflow graph is a language-independent representation of the code in your model. You can build a dataflow graph in Python, store it in a SavedModel (https://www.tensorflow.org/guide/saved_model), and restore it in a C++ program for low-latency inference.

What is a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) ?

A `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) contains two relevant kinds of information:

- **Graph structure.** The nodes and edges of the graph, indicating how individual operations are composed together, but not prescribing how they should be used. The graph structure is like assembly code: inspecting it can convey some useful information, but it does not contain all of the useful context that source code conveys.
- **Graph collections.** TensorFlow provides a general mechanism for storing collections of metadata in a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph). The `tf.add_to_collection` (https://www.tensorflow.org/api_docs/python/tf/add_to_collection) function enables you to associate a list of objects with a key (where `tf.GraphKeys` (https://www.tensorflow.org/api_docs/python/tf/GraphKeys) defines some of the standard keys), and `tf.get_collection` (https://www.tensorflow.org/api_docs/python/tf/get_collection) enables you to look up all objects associated with a key. Many parts of the TensorFlow library use this facility: for example, when you create a `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable), it is added by default to collections representing "global variables" and "trainable variables". When you

later come to create a `tf.train.Saver` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) or `tf.train.Optimizer` (https://www.tensorflow.org/api_docs/python/tf/train/Optimizer), the variables in these collections are used as the default arguments.

Building a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph)

Most TensorFlow programs start with a dataflow graph construction phase. In this phase, you invoke TensorFlow API functions that construct new `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) (node) and `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) (edge) objects and add them to a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) instance. TensorFlow provides a **default graph** that is an implicit argument to all API functions in the same context. For example:

- Calling `tf.constant(42.0)` (https://www.tensorflow.org/api_docs/python/tf/constant) creates a single `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that produces the value `42.0`, adds it to the default graph, and returns a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) that represents the value of the constant.
- Calling `tf.matmul(x, y)` (https://www.tensorflow.org/api_docs/python/tf/linalg/matmul) creates a single `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that multiplies the values of `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects `x` and `y`, adds it to the default graph, and returns a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) that represents the result of the multiplication.
- Executing `v = tf.Variable(0)` adds to the graph a `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that will store a writeable tensor value that persists between `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) calls. The `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) object wraps this operation, and can be used like a tensor (#tensor_like_objects), which

will read the current value of the stored value. The `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) object also has methods such as `tf.Variable.assign` (https://www.tensorflow.org/api_docs/python/tf/Variable#assign) and `tf.Variable.assign_add` (https://www.tensorflow.org/api_docs/python/tf/Variable#assign_add) that create `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) objects that, when executed, update the stored value. (See Variables (https://www.tensorflow.org/guide/variables) for more information about variables.)

- Calling `tf.train.Optimizer.minimize` (https://www.tensorflow.org/api_docs/python/tf/train/Optimizer#minimize) will add operations and tensors to the default graph that calculates gradients, and return a `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that, when run, will apply those gradients to a set of variables.

Most programs rely solely on the default graph. However, see Dealing with multiple graphs (#programming_with_multiple_graphs) for more advanced use cases. High-level APIs such as the `tf.estimator.Estimator` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) API manage the default graph on your behalf, and--for example--may create different graphs for training and evaluation.

Note: Calling most functions in the TensorFlow API merely adds operations and tensors to the default graph, but **does not** perform the actual computation. Instead, you compose these functions until you have a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) or `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that represents the overall computation--such as performing one step of gradient descent--and then pass that object to a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) to perform the computation. See the section "Executing a graph in a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session)" for more details.

Naming operations

A **`tf.Graph`** (https://www.tensorflow.org/api_docs/python/tf/Graph) object defines a **namespace** for the **`tf.Operation`** (https://www.tensorflow.org/api_docs/python/tf/Operation) objects it contains. TensorFlow automatically chooses a unique name for each operation in your graph, but giving operations descriptive names can make your program easier to read and debug. The TensorFlow API provides two ways to override the name of an operation:

- Each API function that creates a new **`tf.Operation`** (https://www.tensorflow.org/api_docs/python/tf/Operation) or returns a new **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) accepts an optional **`name`** argument. For example, **`tf.constant(42.0, name="answer")`** creates a new **`tf.Operation`** named "**`answer`**" and returns a **`tf.Tensor`** named "**`answer:0`**". If the default graph already contains an operation named "**`answer`**", then TensorFlow would append "**`_1`**", "**`_2`**", and so on to the name, in order to make it unique.
- The **`tf.name_scope`** (https://www.tensorflow.org/api_docs/python/tf/name_scope) function makes it possible to add a **name scope** prefix to all operations created in a particular context. The current name scope prefix is a "**`/`**"-delimited list of the names of all active **`tf.name_scope`** (https://www.tensorflow.org/api_docs/python/tf/name_scope) context managers. If a name scope has already been used in the current context, TensorFlow appends "**`_1`**", "**`_2`**", and so on. For example:

```
c_0 = tf.constant(0, name="c")  # => operation named "c"

# Already-used names will be "uniquified".
c_1 = tf.constant(2, name="c")  # => operation named "c_1"

# Name scopes add a prefix to all operations created in the same con
with tf.name_scope("outer"):
  c_2 = tf.constant(2, name="c")  # => operation named "outer/c"

  # Name scopes nest like paths in a hierarchical file system.
  with tf.name_scope("inner"):
    c_3 = tf.constant(3, name="c")  # => operation named "outer/inne

  # Exiting a name scope context will return to the previous prefix.
  c_4 = tf.constant(4, name="c")  # => operation named "outer/c_1"
```

```
# Already-used name scopes will be "uniquified".
with tf.name_scope("inner"):
  c_5 = tf.constant(5, name="c")  # => operation named "outer/inner
```

The graph visualizer uses name scopes to group operations and reduce the visual complexity of a graph. See Visualizing your graph (#visualizing_your_graph) for more information.

Note that `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects are implicitly named after the `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) that produces the tensor as output. A tensor name has the form "`<OP_NAME>:<i>`" where:

- "`<OP_NAME>`" is the name of the operation that produces it.
- "`<i>`" is an integer representing the index of that tensor among the operation's outputs.

Placing operations on different devices

If you want your TensorFlow program to use multiple different devices, the `tf.device` (https://www.tensorflow.org/api_docs/python/tf/device) function provides a convenient way to request that all operations created in a particular context are placed on the same device (or type of device).

A **device specification** has the following form:

```
/job:<JOB_NAME>/task:<TASK_INDEX>/device:<DEVICE_TYPE>:<DEVICE_INDEX>
```

where:

- `<JOB_NAME>` is an alpha-numeric string that does not start with a number.
- `<DEVICE_TYPE>` is a registered device type (such as `GPU` or `CPU`).

- `<TASK_INDEX>` is a non-negative integer representing the index of the task in the job named `<JOB_NAME>`. See `tf.train.ClusterSpec` for an explanation of jobs and tasks.
- `<DEVICE_INDEX>` is a non-negative integer representing the index of the device, for example, to distinguish between different GPU devices used in the same process.

You do not need to specify every part of a device specification. For example, if you are running in a single-machine configuration with a single GPU, you might use `tf.device` (https://www.tensorflow.org/api_docs/python/tf/device) to pin some operations to the CPU and GPU:

```
# Operations created outside either context will run on the "best possible"
# device. For example, if you have a GPU and a CPU available, and the operation
# has a GPU implementation, TensorFlow will choose the GPU.
weights = tf.random_normal(...)

with tf.device("/device:CPU:0"):
  # Operations created in this context will be pinned to the CPU.
  img = tf.decode_jpeg(tf.read_file("img.jpg"))

with tf.device("/device:GPU:0"):
  # Operations created in this context will be pinned to the GPU.
  result = tf.matmul(weights, img)
```

If you are deploying TensorFlow in a typical distributed configuration (https://www.tensorflow.org/deploy/distributed), you might specify the job name and task ID to place variables on a task in the parameter server job ("`/job:ps`"), and the other operations on task in the worker job ("`/job:worker`"):

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable(tf.truncated_normal([784, 100]))
  biases_1 = tf.Variable(tf.zeros([100]))

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable(tf.truncated_normal([100, 10]))
  biases_2 = tf.Variable(tf.zeros([10]))

with tf.device("/job:worker"):
```

```
  layer_1 = tf.matmul(train_batch, weights_1) + biases_1
  layer_2 = tf.matmul(train_batch, weights_2) + biases_2
```

`tf.device` (https://www.tensorflow.org/api_docs/python/tf/device) gives you a lot of flexibility to choose placements for individual operations or broad regions of a TensorFlow graph. In many cases, there are simple heuristics that work well. For example, the `tf.train.replica_device_setter` (https://www.tensorflow.org/api_docs/python/tf/train/replica_device_setter) API can be used with `tf.device` (https://www.tensorflow.org/api_docs/python/tf/device) to place operations for **data-parallel distributed training**. For example, the following code fragment shows how `tf.train.replica_device_setter` (https://www.tensorflow.org/api_docs/python/tf/train/replica_device_setter) applies different placement policies to `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) objects and other operations:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=3)):
  # tf.Variable objects are, by default, placed on tasks in "/job:ps" in a
  # round-robin fashion.
  w_0 = tf.Variable(...)  # placed on "/job:ps/task:0"
  b_0 = tf.Variable(...)  # placed on "/job:ps/task:1"
  w_1 = tf.Variable(...)  # placed on "/job:ps/task:2"
  b_1 = tf.Variable(...)  # placed on "/job:ps/task:0"

  input_data = tf.placeholder(tf.float32)     # placed on "/job:worker"
  layer_0 = tf.matmul(input_data, w_0) + b_0  # placed on "/job:worker"
  layer_1 = tf.matmul(layer_0, w_1) + b_1     # placed on "/job:worker"
```

Tensor-like objects

Many TensorFlow operations take one or more `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects as arguments. For example, `tf.matmul` (https://www.tensorflow.org/api_docs/python/tf/linalg/matmul) takes two `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects, and `tf.add_n` (https://www.tensorflow.org/api_docs/python/tf/math/add_n) takes a list of `n` `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects. For convenience, these functions will accept a **tensor-like object** in place of a

`tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor), and implicitly convert it to a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) using the `tf.convert_to_tensor` (https://www.tensorflow.org/api_docs/python/tf/convert_to_tensor) method. Tensor-like objects include elements of the following types:

- `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor)
- `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable)
- `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html)
- `list` (and lists of tensor-like objects)
- Scalar Python types: `bool`, `float`, `int`, `str`

You can register additional tensor-like types using `tf.register_tensor_conversion_function` (https://www.tensorflow.org/api_docs/python/tf/register_tensor_conversion_function).

Note: By default, TensorFlow will create a new `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) each time you use the same tensor-like object. If the tensor-like object is large (e.g. a `numpy.ndarray` containing a set of training examples) and you use it multiple times, you may run out of memory. To avoid this, manually call `tf.convert_to_tensor` (https://www.tensorflow.org/api_docs/python/tf/convert_to_tensor) on the tensor-like object once and use the returned `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) instead.

Executing a graph in a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session)

TensorFlow uses the `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) class to represent a connection between the client program---typically a Python program, although a similar interface is available in other languages---and the C++ runtime. A `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) object provides access to devices in the local machine, and remote devices using the distributed TensorFlow runtime. It also caches information about your `tf.Graph`

(https://www.tensorflow.org/api_docs/python/tf/Graph) so that you can efficiently run the same computation multiple times.

Creating a `tf.Session`

(https://www.tensorflow.org/api_docs/python/tf/Session)

If you are using the low-level TensorFlow API, you can create a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) for the current default graph as follows:

```
# Create a default in-process session.
with tf.Session() as sess:
  # ...

# Create a remote session.
with tf.Session("grpc://example.org:2222"):
  # ...
```

Since a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) owns physical resources (such as GPUs and network connections), it is typically used as a context manager (in a `with` block) that automatically closes the session when you exit the block. It is also possible to create a session without using a `with` block, but you should explicitly call `tf.Session.close` (https://www.tensorflow.org/api_docs/python/tf/Session#close) when you are finished with it to free the resources.

Note: Higher-level APIs such as `tf.train.MonitoredTrainingSession` (https://www.tensorflow.org/api_docs/python/tf/train/MonitoredTrainingSession) or `tf.estimator.Estimator` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) will create and manage a `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) for you. These APIs accept optional `target` and `config` arguments (either directly, or as part of a `tf.estimator.RunConfig` (https://www.tensorflow.org/api_docs/python/tf/estimator/RunConfig) object), with the same meaning as described below.

`tf.Session.init` (https://www.tensorflow.org/api_docs/python/tf/Session#__init__) accepts three optional arguments:

- **`target`.** If this argument is left empty (the default), the session will only use devices in the local machine. However, you may also specify a `grpc://` URL to specify the address of a TensorFlow server, which gives the session access to all devices on machines that this server controls. See `tf.train.Server` (https://www.tensorflow.org/api_docs/python/tf/distribute/Server) for details of how to create a TensorFlow server. For example, in the common **between-graph replication** configuration, the `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) connects to a `tf.train.Server` (https://www.tensorflow.org/api_docs/python/tf/distribute/Server) in the same process as the client. The distributed TensorFlow (https://www.tensorflow.org/deploy/distributed) deployment guide describes other common scenarios.

- **`graph`.** By default, a new `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session) will be bound to---and only able to run operations in---the current default graph. If you are using multiple graphs in your program (see Programming with multiple graphs (#programming_with_multiple_graphs) for more details), you can specify an explicit `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) when you construct the session.

- **`config`.** This argument allows you to specify a `tf.ConfigProto` (https://www.tensorflow.org/api_docs/python/tf/ConfigProto) that controls the behavior of the session. For example, some of the configuration options include:

    - `allow_soft_placement`. Set this to `True` to enable a "soft" device placement algorithm, which ignores `tf.device` (https://www.tensorflow.org/api_docs/python/tf/device) annotations that attempt to place CPU-only operations on a GPU device, and places them on the CPU instead.

    - `cluster_def`. When using distributed TensorFlow, this option allows you to specify what machines to use in the computation, and provide a mapping between job names, task indices, and network addresses. See `tf.train.ClusterSpec.as_cluster_def` (https://www.tensorflow.org/api_docs/python/tf/train/ClusterSpec#as_cluster_def) for details.

- `graph_options.optimizer_options`. Provides control over the optimizations that TensorFlow performs on your graph before executing it.
- `gpu_options.allow_growth`. Set this to `True` to change the GPU memory allocator so that it gradually increases the amount of memory allocated, rather than allocating most of the memory at startup.

Using `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) to execute operations

The `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) method is the main mechanism for running a `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) or evaluating a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor). You can pass one or more `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) or `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects to `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run), and TensorFlow will execute the operations that are needed to compute the result.

`tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) requires you to specify a list of **fetches**, which determine the return values, and may be a `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation), a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor), or a tensor-like type (#tensor_like_objects) such as `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable). These fetches determine what **subgraph** of the overall `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) must be executed to produce the result: this is the subgraph that contains all operations named in the fetch list, plus all operations whose outputs are used to compute the value of the fetches. For example, the following code fragment shows how different arguments to `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) cause different subgraphs to be executed:

```
x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
w = tf.Variable(tf.random_uniform([2, 2]))
```

```
y = tf.matmul(x, w)
output = tf.nn.softmax(y)
init_op = w.initializer

with tf.Session() as sess:
  # Run the initializer on `w`.
  sess.run(init_op)

  # Evaluate `output`. `sess.run(output)` will return a NumPy array containing
  # the result of the computation.
  print(sess.run(output))

  # Evaluate `y` and `output`. Note that `y` will only be computed once, and its
  # result used both to return `y_val` and as an input to the `tf.nn.softmax()`
  # op. Both `y_val` and `output_val` will be NumPy arrays.
  y_val, output_val = sess.run([y, output])
```

`tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) also optionally takes a dictionary of **feeds**, which is a mapping from `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects (typically `tf.placeholder` (https://www.tensorflow.org/api_docs/python/tf/placeholder) tensors) to values (typically Python scalars, lists, or NumPy arrays) that will be substituted for those tensors in the execution. For example:

```
# Define a placeholder that expects a vector of three floating-point values,
# and a computation that depends on it.
x = tf.placeholder(tf.float32, shape=[3])
y = tf.square(x)

with tf.Session() as sess:
  # Feeding a value changes the result that is returned when you evaluate `y`.
  print(sess.run(y, {x: [1.0, 2.0, 3.0]}))  # => "[1.0, 4.0, 9.0]"
  print(sess.run(y, {x: [0.0, 0.0, 5.0]}))  # => "[0.0, 0.0, 25.0]"

  # Raises `tf.errors.InvalidArgumentError`, because you must feed a value for
  # a `tf.placeholder()` when evaluating a tensor that depends on it.
  sess.run(y)

  # Raises `ValueError`, because the shape of `37.0` does not match the shape
  # of placeholder `x`.
  sess.run(y, {x: 37.0})
```

`tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run) also accepts an optional `options` argument that enables you to specify options about the call, and an optional `run_metadata` argument that enables you to collect metadata about the execution. For example, you can use these options together to collect tracing information about the execution:

```
y = tf.matmul([[37.0, -23.0], [1.0, 4.0]], tf.random_uniform([2, 2]))

with tf.Session() as sess:
  # Define options for the `sess.run()` call.
  options = tf.RunOptions()
  options.output_partition_graphs = True
  options.trace_level = tf.RunOptions.FULL_TRACE

  # Define a container for the returned metadata.
  metadata = tf.RunMetadata()

  sess.run(y, options=options, run_metadata=metadata)

  # Print the subgraphs that executed on each device.
  print(metadata.partition_graphs)

  # Print the timings of each operation that executed.
  print(metadata.step_stats)
```

Visualizing your graph

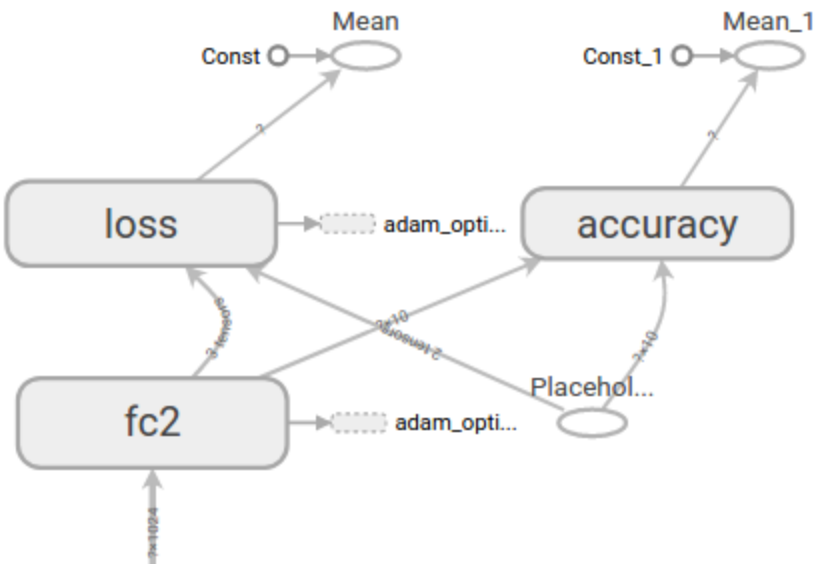
TensorFlow includes tools that can help you to understand the code in a graph. The **graph visualizer** is a component of TensorBoard that renders the structure of your graph visually in a browser. The easiest way to create a visualization is to pass a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) when creating the `tf.summary.FileWriter` (https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter):

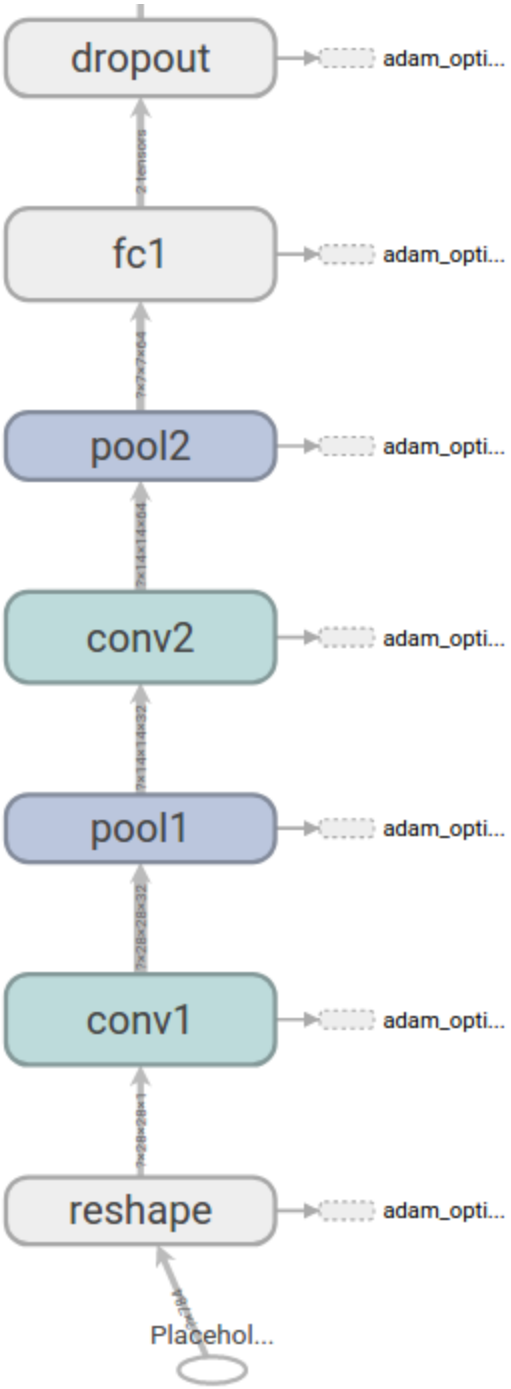
```
# Build your graph.
x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
w = tf.Variable(tf.random_uniform([2, 2]))
```

```
y = tf.matmul(x, w)
# ...
loss = ...
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

with tf.Session() as sess:
  # `sess.graph` provides access to the graph used in a `tf.Session`.
  writer = tf.summary.FileWriter("/tmp/log/...", sess.graph)

  # Perform your computation...
  for i in range(1000):
    sess.run(train_op)
    # ...

  writer.close()
```

Note: If you are using a **`tf.estimator.Estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator), the graph (and any summaries) will be logged automatically to the **`model_dir`** that you specified when creating the estimator.

> You can then open the log in **`tensorboard`**, navigate to the "Graph" tab, and see a high-level visualization of your graph's structure. Note that a typical TensorFlow graph---especially training graphs with automatically computed gradients---has too many nodes to visualize at once. The graph visualizer makes use of name scopes to group related operations into "super" nodes. You can click on the orange "+" button on any of these super nodes to expand the subgraph inside.





For more information about visualizing your TensorFlow application with TensorBoard, see the TensorBoard guide (https://www.tensorflow.org/guide/summaries_and_tensorboard).

Programming with multiple graphs

Note: When training a model, a common way of organizing your code is to use one graph for training your model, and a separate graph for evaluating or performing inference with a trained model. In many cases, the inference graph will be different from the training graph: for example, techniques like dropout and batch normalization use different operations in each case. Furthermore, by default utilities like **`tf.train.Saver`** (https://www.tensorflow.org/api_docs/python/tf/train/Saver) use the names of **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) objects (which have names based on an underlying **`tf.Operation`** (https://www.tensorflow.org/api_docs/python/tf/Operation)) to identify each variable in a saved checkpoint. When programming this way, you can either use completely separate Python processes to build and execute the graphs, or you can use multiple graphs in the same process. This section describes how to use multiple graphs in the same process.

As noted above, TensorFlow provides a "default graph" that is implicitly passed to all API functions in the same context. For many applications, a single graph is sufficient. However, TensorFlow also provides methods for manipulating the default graph, which can be useful in more advanced use cases. For example:

- A `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) defines the namespace for `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) objects: each operation in a single graph must have a unique name. TensorFlow will "uniquify" the names of operations by appending `"_1"`, `"_2"`, and so on to their names if the requested name is already taken. Using multiple explicitly created graphs gives you more control over what name is given to each operation.
- The default graph stores information about every `tf.Operation` (https://www.tensorflow.org/api_docs/python/tf/Operation) and `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) that was ever added to it. If your program creates a large number of unconnected subgraphs, it may be more efficient to use a different `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) to build each subgraph, so that unrelated state can be garbage collected.

You can install a different `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) as the default graph, using the `tf.Graph.as_default` (https://www.tensorflow.org/api_docs/python/tf/Graph#as_default) context manager:

```
g_1 = tf.Graph()
with g_1.as_default():
  # Operations created in this scope will be added to `g_1`.
  c = tf.constant("Node in g_1")

  # Sessions created in this scope will run operations from `g_1`.
  sess_1 = tf.Session()

g_2 = tf.Graph()
with g_2.as_default():
  # Operations created in this scope will be added to `g_2`.
  d = tf.constant("Node in g_2")

# Alternatively, you can pass a graph when constructing a `tf.Session`:
# `sess_2` will run operations from `g_2`.
sess_2 = tf.Session(graph=g_2)

assert c.graph is g_1
assert sess_1.graph is g_1

assert d.graph is g_2
assert sess_2.graph is g_2
```

To inspect the current default graph, call **`tf.get_default_graph`** (https://www.tensorflow.org/api_docs/python/tf/get_default_graph), which returns a **`tf.Graph`** (https://www.tensorflow.org/api_docs/python/tf/Graph) object:

```
# Print all of the operations in the default graph.
g = tf.get_default_graph()
print(g.get_operations())
```