# Programmer's Guide

The documents in this unit dive into the details of writing TensorFlow code. This section begins with the following guides, each of which explain a particular aspect of TensorFlow:

* [Variables: Creation, Initialization, Saving, and Loading](https://www.tensorflow.org/programmers_guide/variables), which details the mechanics of TensorFlow Variables.
* [Tensor Ranks, Shapes, and Types](https://www.tensorflow.org/programmers_guide/dims_types), which explains Tensor rank (the number of dimensions), shape (the size of each dimension), and datatypes.
* [Sharing Variables](https://www.tensorflow.org/programmers_guide/variable_scope), which explains how to share and manage large sets of variables when building complex models.
* [Threading and Queues](https://www.tensorflow.org/programmers_guide/threading_and_queues), which explains TensorFlow's rich queuing system.
* [Reading Data](https://www.tensorflow.org/programmers_guide/reading_data), which documents three different mechanisms for getting data into a TensorFlow program.

The following guide is helpful when training a complex model over multiple days:

* [Supervisor: Training Helper for Days-Long Trainings](https://www.tensorflow.org/programmers_guide/supervisor), which explains how to gracefully handle system crashes during a lengthy training session.

TensorFlow provides a debugger named tfdbg, which is documented in the following two guides:

* [TensorFlow Debugger (tfdbg) Command-Line-Interface Tutorial: MNIST](https://www.tensorflow.org/programmers_guide/debugger), which walks you through the use of tfdbg within an application written in the low-level TensorFlow API.
* [How to Use TensorFlow Debugger (tfdbg) with tf.contrib.learn](https://www.tensorflow.org/programmers_guide/tfdbg-tflearn), which demonstrates how to use tfdbgwithin the Estimators API.

A MetaGraph consists of both a computational graph and its associated metadata. A MetaGraph contains the information required to continue training, perform evaluation, or run inference on a previously trained graph. The following guide details MetaGraph objects:

* [Exporting and Importing a MetaGraph](https://www.tensorflow.org/programmers_guide/meta_graph).

To learn about the TensorFlow versioning scheme, consult the following two guides:

* [TensorFlow Version Semantics](https://www.tensorflow.org/programmers_guide/version_semantics), which explains TensorFlow's versioning nomenclature and compatibility rules.
* [TensorFlow Data Versioning: GraphDefs and Checkpoints](https://www.tensorflow.org/programmers_guide/data_versions), which explains how TensorFlow adds versioning information to computational graphs and checkpoints in order to support compatibility across versions.

We conclude this section with a FAQ about TensorFlow programming:

* [Frequently Asked Questions](https://www.tensorflow.org/programmers_guide/faq)

# Variables: Creation, Initialization, Saving, and Loading

When you train a model, you use [variables](https://www.tensorflow.org/api_guides/python/state_ops) to hold and update parameters. Variables are in-memory buffers containing tensors. They must be explicitly initialized and can be saved to disk during and after training. You can later restore saved values to exercise or analyze the model.

This document references the following TensorFlow classes. Follow the links to their reference manual for a complete description of their API:

* The [tf.Variable](https://www.tensorflow.org/api_docs/python/tf/Variable) class.
* The [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver) class.

## Creation

When you create a [Variable](https://www.tensorflow.org/api_guides/python/state_ops) you pass a Tensor as its initial value to the Variable() constructor. TensorFlow provides a collection of ops that produce tensors often used for initialization from [constants or random values](https://www.tensorflow.org/api_guides/python/constant_op).

Note that all these ops require you to specify the shape of the tensors. That shape automatically becomes the shape of the variable. Variables generally have a fixed shape, but TensorFlow provides advanced mechanisms to reshape variables.

# Create two variables.  
weights = tf.Variable(tf.random\_normal([784, 200], stddev=0.35),  
                      name="weights")  
biases = tf.Variable(tf.zeros([200]), name="biases")

Calling tf.Variable() adds several ops to the graph:

* A variable op that holds the variable value.
* An initializer op that sets the variable to its initial value. This is actually a tf.assign op.
* The ops for the initial value, such as the zeros op for the biases variable in the example are also added to the graph.

The value returned by tf.Variable() value is an instance of the Python class tf.Variable.

### Device placement

A variable can be pinned to a particular device when it is created, using a [with tf.device(...):](https://www.tensorflow.org/api_docs/python/tf/device)block:

# Pin a variable to CPU.  
with tf.device("/cpu:0"):  
  v = tf.Variable(...)  
  
# Pin a variable to GPU.  
with tf.device("/gpu:0"):  
  v = tf.Variable(...)  
  
# Pin a variable to a particular parameter server task.  
with tf.device("/job:ps/task:7"):  
  v = tf.Variable(...)

**N.B.** Operations that mutate a variable, such as [tf.Variable.assign](https://www.tensorflow.org/api_docs/python/tf/Variable#assign) and the parameter update operations in a [tf.train.Optimizer](https://www.tensorflow.org/api_docs/python/tf/train/Optimizer) must run on the same device as the variable. Incompatible device placement directives will be ignored when creating these operations.

Device placement is particularly important when running in a replicated setting. See[tf.train.replica\_device\_setter](https://www.tensorflow.org/api_docs/python/tf/train/replica_device_setter) for details of a device function that can simplify the configuration for devices for a replicated model.

## Initialization

Variable initializers must be run explicitly before other ops in your model can be run. The easiest way to do that is to add an op that runs all the variable initializers, and run that op before using the model.

You can alternatively restore variable values from a checkpoint file, see below.

Use tf.global\_variables\_initializer() to add an op to run variable initializers. Only run that op after you have fully constructed your model and launched it in a session.

# Create two variables.  
weights = tf.Variable(tf.random\_normal([784, 200], stddev=0.35),  
                      name="weights")  
biases = tf.Variable(tf.zeros([200]), name="biases")  
...  
# Add an op to initialize the variables.  
init\_op = tf.global\_variables\_initializer()  
  
# Later, when launching the model  
with tf.Session() as sess:  
  # Run the init operation.  
  sess.run(init\_op)  
  ...  
  # Use the model  
  ...

### Initialization from another Variable

You sometimes need to initialize a variable from the initial value of another variable. As the op added by tf.global\_variables\_initializer() initializes all variables in parallel you have to be careful when this is needed.

To initialize a new variable from the value of another variable use the other variable's initialized\_value() property. You can use the initialized value directly as the initial value for the new variable, or you can use it as any other tensor to compute a value for the new variable.

# Create a variable with a random value.  
weights = tf.Variable(tf.random\_normal([784, 200], stddev=0.35),  
                      name="weights")  
# Create another variable with the same value as 'weights'.  
w2 = tf.Variable(weights.initialized\_value(), name="w2")  
# Create another variable with twice the value of 'weights'  
w\_twice = tf.Variable(weights.initialized\_value() \* 2.0, name="w\_twice")

### Custom Initialization

The convenience function tf.global\_variables\_initializer() adds an op to initialize all variables in the model. You can also pass an explicit list of variables to initialize to tf.variables\_initializer. See the [Variables Documentation](https://www.tensorflow.org/api_guides/python/state_ops) for more options, including checking if variables are initialized.

## Saving and Restoring

The easiest way to save and restore a model is to use a tf.train.Saver object. The constructor adds save and restore ops to the graph for all, or a specified list, of the variables in the graph. The saver object provides methods to run these ops, specifying paths for the checkpoint files to write to or read from.

### Checkpoint Files

Variables are saved in binary files that, roughly, contain a map from variable names to tensor values.

When you create a Saver object, you can optionally choose names for the variables in the checkpoint files. By default, it uses the value of the [tf.Variable.name](https://www.tensorflow.org/api_docs/python/tf/Variable#name) property for each variable.

To understand what variables are in a checkpoint, you can use the [inspect\_checkpoint](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/python/tools/inspect_checkpoint.py) library, and in particular, the print\_tensors\_in\_checkpoint\_file function.

### Saving Variables

Create a Saver with tf.train.Saver() to manage all variables in the model.

# Create some variables.  
v1 = tf.Variable(..., name="v1")  
v2 = tf.Variable(..., name="v2")  
...  
# Add an op to initialize the variables.  
init\_op = tf.global\_variables\_initializer()  
  
# Add ops to save and restore all the variables.  
saver = tf.train.Saver()  
  
# Later, launch the model, initialize the variables, do some work, save the  
# variables to disk.  
with tf.Session() as sess:  
  sess.run(init\_op)  
  # Do some work with the model.  
  ..  
  # Save the variables to disk.  
  save\_path = saver.save(sess, "/tmp/model.ckpt")  
  print("Model saved in file: %s" % save\_path)

### Restoring Variables

The same Saver object is used to restore variables. Note that when you restore variables from a file you do not have to initialize them beforehand.

# Create some variables.  
v1 = tf.Variable(..., name="v1")  
v2 = tf.Variable(..., name="v2")  
...  
# Add ops to save and restore all the variables.  
saver = tf.train.Saver()  
  
# Later, launch the model, use the saver to restore variables from disk, and  
# do some work with the model.  
with tf.Session() as sess:  
  # Restore variables from disk.  
  saver.restore(sess, "/tmp/model.ckpt")  
  print("Model restored.")  
  # Do some work with the model  
  ...

### Choosing which Variables to Save and Restore

If you do not pass any argument to tf.train.Saver() the saver handles all variables in the graph. Each one of them is saved under the name that was passed when the variable was created.

It is sometimes useful to explicitly specify names for variables in the checkpoint files. For example, you may have trained a model with a variable named "weights" whose value you want to restore in a new variable named "params".

It is also sometimes useful to only save or restore a subset of the variables used by a model. For example, you may have trained a neural net with 5 layers, and you now want to train a new model with 6 layers, restoring the parameters from the 5 layers of the previously trained model into the first 5 layers of the new model.

You can easily specify the names and variables to save by passing to the tf.train.Saver()constructor a Python dictionary: keys are the names to use, values are the variables to manage.

Notes:

You can create as many saver objects as you want if you need to save and restore different subsets of the model variables. The same variable can be listed in multiple saver objects, its value is only changed when the saver restore() method is run.

If you only restore a subset of the model variables at the start of a session, you have to run an initialize op for the other variables. See [tf.variables\_initializer](https://www.tensorflow.org/api_docs/python/tf/variables_initializer) for more information.

# Create some variables.  
v1 = tf.Variable(..., name="v1")  
v2 = tf.Variable(..., name="v2")  
...  
# Add ops to save and restore only 'v2' using the name "my\_v2"  
saver = tf.train.Saver({"my\_v2": v2})  
# Use the saver object normally after that.  
...

# Tensor Ranks, Shapes, and Types

TensorFlow programs use a tensor data structure to represent all data. You can think of a TensorFlow tensor as an n-dimensional array or list. A tensor has a static type and dynamic dimensions. Only tensors may be passed between nodes in the computation graph.

## Rank

In the TensorFlow system, tensors are described by a unit of dimensionality known as rank. Tensor rank is not the same as matrix rank. Tensor rank (sometimes referred to as order or degree or n-dimension) is the number of dimensions of the tensor. For example, the following tensor (defined as a Python list) has a rank of 2:

t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

A rank two tensor is what we typically think of as a matrix, a rank one tensor is a vector. For a rank two tensor you can access any element with the syntax t[i, j]. For a rank three tensor you would need to address an element with t[i, j, k].

| Rank | Math entity | Python example |
| --- | --- | --- |
| 0 | Scalar (magnitude only) | s = 483 |
| 1 | Vector (magnitude and direction) | v = [1.1, 2.2, 3.3] |
| 2 | Matrix (table of numbers) | m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] |
| 3 | 3-Tensor (cube of numbers) | t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16],[18]]] |
| n | n-Tensor (you get the idea) | .... |

## Shape

The TensorFlow documentation uses three notational conventions to describe tensor dimensionality: rank, shape, and dimension number. The following table shows how these relate to one another:

| Rank | Shape | Dimension number | Example |
| --- | --- | --- | --- |
| 0 | [] | 0-D | A 0-D tensor. A scalar. |
| 1 | [D0] | 1-D | A 1-D tensor with shape [5]. |
| 2 | [D0, D1] | 2-D | A 2-D tensor with shape [3, 4]. |
| 3 | [D0, D1, D2] | 3-D | A 3-D tensor with shape [1, 4, 3]. |
| n | [D0, D1, ... Dn-1] | n-D | A tensor with shape [D0, D1, ... Dn-1]. |

Shapes can be represented via Python lists / tuples of ints, or with the [tf.TensorShape](https://www.tensorflow.org/api_docs/python/tf/TensorShape).

## Data types

In addition to dimensionality, Tensors have a data type. You can assign any one of the following data types to a tensor:

| Data type | Python type | Description |
| --- | --- | --- |
| DT\_FLOAT | tf.float32 | 32 bits floating point. |
| DT\_DOUBLE | tf.float64 | 64 bits floating point. |
| DT\_INT8 | tf.int8 | 8 bits signed integer. |
| DT\_INT16 | tf.int16 | 16 bits signed integer. |
| DT\_INT32 | tf.int32 | 32 bits signed integer. |
| DT\_INT64 | tf.int64 | 64 bits signed integer. |
| DT\_UINT8 | tf.uint8 | 8 bits unsigned integer. |
| DT\_UINT16 | tf.uint16 | 16 bits unsigned integer. |
| DT\_STRING | tf.string | Variable length byte arrays. Each element of a Tensor is a byte array. |
| DT\_BOOL | tf.bool | Boolean. |
| DT\_COMPLEX64 | tf.complex64 | Complex number made of two 32 bits floating points: real and imaginary parts. |
| DT\_COMPLEX128 | tf.complex128 | Complex number made of two 64 bits floating points: real and imaginary parts. |
| DT\_QINT8 | tf.qint8 | 8 bits signed integer used in quantized Ops. |
| DT\_QINT32 | tf.qint32 | 32 bits signed integer used in quantized Ops. |
| DT\_QUINT8 | tf.quint8 | 8 bits unsigned integer used in quantized Ops. |

# Sharing Variables

You can create, initialize, save and load single variables in the way described in the [Variables HowTo](https://www.tensorflow.org/programmers_guide/variables). But when building complex models you often need to share large sets of variables and you might want to initialize all of them in one place. This tutorial shows how this can be done using tf.variable\_scope() and the tf.get\_variable().

## The Problem

Imagine you create a simple model for image filters, similar to our [Convolutional Neural Networks Tutorial](https://www.tensorflow.org/tutorials/deep_cnn) model but with only 2 convolutions (for simplicity of this example). If you use just tf.Variable, as explained in [Variables HowTo](https://www.tensorflow.org/programmers_guide/variables), your model might look like this.

def my\_image\_filter(input\_images):  
    conv1\_weights = tf.Variable(tf.random\_normal([5, 5, 32, 32]),  
        name="conv1\_weights")  
    conv1\_biases = tf.Variable(tf.zeros([32]), name="conv1\_biases")  
    conv1 = tf.nn.conv2d(input\_images, conv1\_weights,  
        strides=[1, 1, 1, 1], padding='SAME')  
    relu1 = tf.nn.relu(conv1 + conv1\_biases)  
  
    conv2\_weights = tf.Variable(tf.random\_normal([5, 5, 32, 32]),  
        name="conv2\_weights")  
    conv2\_biases = tf.Variable(tf.zeros([32]), name="conv2\_biases")  
    conv2 = tf.nn.conv2d(relu1, conv2\_weights,  
        strides=[1, 1, 1, 1], padding='SAME')  
    return tf.nn.relu(conv2 + conv2\_biases)

As you can easily imagine, models quickly get much more complicated than this one, and even here we already have 4 different variables: conv1\_weights, conv1\_biases, conv2\_weights, and conv2\_biases.

The problem arises when you want to reuse this model. Assume you want to apply your image filter to 2 different images, image1 and image2. You want both images processed by the same filter with the same parameters. You can call my\_image\_filter() twice, but this will create two sets of variables, 4 variables in each one, for a total of 8 variables.

# First call creates one set of 4 variables.  
result1 = my\_image\_filter(image1)  
# Another set of 4 variables is created in the second call.  
result2 = my\_image\_filter(image2)

A common way to share variables is to create them in a separate piece of code and pass them to functions that use them. For example by using a dictionary:

variables\_dict = {  
    "conv1\_weights": tf.Variable(tf.random\_normal([5, 5, 32, 32]),  
        name="conv1\_weights")  
    "conv1\_biases": tf.Variable(tf.zeros([32]), name="conv1\_biases")  
    ... etc. ...  
}  
  
def my\_image\_filter(input\_images, variables\_dict):  
    conv1 = tf.nn.conv2d(input\_images, variables\_dict["conv1\_weights"],  
        strides=[1, 1, 1, 1], padding='SAME')  
    relu1 = tf.nn.relu(conv1 + variables\_dict["conv1\_biases"])  
  
    conv2 = tf.nn.conv2d(relu1, variables\_dict["conv2\_weights"],  
        strides=[1, 1, 1, 1], padding='SAME')  
    return tf.nn.relu(conv2 + variables\_dict["conv2\_biases"])  
  
# Both calls to my\_image\_filter() now use the same variables  
result1 = my\_image\_filter(image1, variables\_dict)  
result2 = my\_image\_filter(image2, variables\_dict)

While convenient, creating variables like above, outside of the code, breaks encapsulation:

* The code that builds the graph must document the names, types, and shapes of variables to create.
* When the code changes, the callers may have to create more, or less, or different variables.

One way to address the problem is to use classes to create a model, where the classes take care of managing the variables they need. For a lighter solution, not involving classes, TensorFlow provides a Variable Scope mechanism that allows to easily share named variables while constructing a graph.

## Variable Scope Example

Variable Scope mechanism in TensorFlow consists of two main functions:

* tf.get\_variable(<name>, <shape>, <initializer>): Creates or returns a variable with a given name.
* tf.variable\_scope(<scope\_name>): Manages namespaces for names passed to tf.get\_variable().

The function tf.get\_variable() is used to get or create a variable instead of a direct call to tf.Variable. It uses an initializer instead of passing the value directly, as in tf.Variable. An initializer is a function that takes the shape and provides a tensor with that shape. Here are some initializers available in TensorFlow:

* tf.constant\_initializer(value) initializes everything to the provided value,
* tf.random\_uniform\_initializer(a, b) initializes uniformly from [a, b],
* tf.random\_normal\_initializer(mean, stddev) initializes from the normal distribution with the given mean and standard deviation.

To see how tf.get\_variable() solves the problem discussed before, let's refactor the code that created one convolution into a separate function, named conv\_relu:

def conv\_relu(input, kernel\_shape, bias\_shape):  
    # Create variable named "weights".  
    weights = tf.get\_variable("weights", kernel\_shape,  
        initializer=tf.random\_normal\_initializer())  
    # Create variable named "biases".  
    biases = tf.get\_variable("biases", bias\_shape,  
        initializer=tf.constant\_initializer(0.0))  
    conv = tf.nn.conv2d(input, weights,  
        strides=[1, 1, 1, 1], padding='SAME')  
    return tf.nn.relu(conv + biases)

This function uses short names "weights" and "biases". We'd like to use it for both conv1 and conv2, but the variables need to have different names. This is where tf.variable\_scope() comes into play: it pushes a namespace for variables.

def my\_image\_filter(input\_images):  
    with tf.variable\_scope("conv1"):  
        # Variables created here will be named "conv1/weights", "conv1/biases".  
        relu1 = conv\_relu(input\_images, [5, 5, 32, 32], [32])  
    with tf.variable\_scope("conv2"):  
        # Variables created here will be named "conv2/weights", "conv2/biases".  
        return conv\_relu(relu1, [5, 5, 32, 32], [32])

Now, let's see what happens when we call my\_image\_filter() twice.

result1 = my\_image\_filter(image1)  
result2 = my\_image\_filter(image2)  
# Raises ValueError(... conv1/weights already exists ...)

As you can see, tf.get\_variable() checks that already existing variables are not shared by accident. If you want to share them, you need to specify it by setting reuse\_variables() as follows.

with tf.variable\_scope("image\_filters") as scope:  
    result1 = my\_image\_filter(image1)  
    scope.reuse\_variables()  
    result2 = my\_image\_filter(image2)

This is a good way to share variables, lightweight and safe.

## How Does Variable Scope Work?

### Understanding tf.get\_variable()

To understand variable scope it is necessary to first fully understand how tf.get\_variable()works. Here is how tf.get\_variable is usually called.

v = tf.get\_variable(name, shape, dtype, initializer)

This call does one of two things depending on the scope it is called in. Here are the two options.

* Case 1: the scope is set for creating new variables, as evidenced bytf.get\_variable\_scope().reuse == False.

In this case, v will be a newly created tf.Variable with the provided shape and data type. The full name of the created variable will be set to the current variable scope name + the provided name and a check will be performed to ensure that no variable with this full name exists yet. If a variable with this full name already exists, the function will raise a ValueError. If a new variable is created, it will be initialized to the value initializer(shape). For example:

with tf.variable\_scope("foo"):  
    v = tf.get\_variable("v", [1])  
assert v.name == "foo/v:0"

* Case 2: the scope is set for reusing variables, as evidenced bytf.get\_variable\_scope().reuse == True.

In this case, the call will search for an already existing variable with name equal to the current variable scope name + the provided name. If no such variable exists, a ValueError will be raised. If the variable is found, it will be returned. For example:

with tf.variable\_scope("foo"):  
    v = tf.get\_variable("v", [1])  
with tf.variable\_scope("foo", reuse=True):  
    v1 = tf.get\_variable("v", [1])  
assert v1 is v

### Basics of tf.variable\_scope()

Knowing how tf.get\_variable() works makes it easy to understand variable scope. The primary function of variable scope is to carry a name that will be used as prefix for variable names and a reuse-flag to distinguish the two cases described above. Nesting variable scopes appends their names in a way analogous to how directories work:

with tf.variable\_scope("foo"):  
    with tf.variable\_scope("bar"):  
        v = tf.get\_variable("v", [1])  
assert v.name == "foo/bar/v:0"

The current variable scope can be retrieved using tf.get\_variable\_scope() and the reuse flag of the current variable scope can be set to True by calling tf.get\_variable\_scope().reuse\_variables():

with tf.variable\_scope("foo"):  
    v = tf.get\_variable("v", [1])  
    tf.get\_variable\_scope().reuse\_variables()  
    v1 = tf.get\_variable("v", [1])  
assert v1 is v

Note that you cannot set the reuse flag to False. The reason behind this is to allow to compose functions that create models. Imagine you write a function my\_image\_filter(inputs) as before. Someone calling the function in a variable scope with reuse=True would expect all inner variables to be reused as well. Allowing to force reuse=False inside the function would break this contract and make it hard to share parameters in this way.

Even though you cannot set reuse to False explicitly, you can enter a reusing variable scope and then exit it, going back to a non-reusing one. This can be done using a reuse=True parameter when opening a variable scope. Note also that, for the same reason as above, the reuse parameter is inherited. So when you open a reusing variable scope, all sub-scopes will be reusing too.

with tf.variable\_scope("root"):  
    # At start, the scope is not reusing.  
    assert tf.get\_variable\_scope().reuse == False  
    with tf.variable\_scope("foo"):  
        # Opened a sub-scope, still not reusing.  
        assert tf.get\_variable\_scope().reuse == False  
    with tf.variable\_scope("foo", reuse=True):  
        # Explicitly opened a reusing scope.  
        assert tf.get\_variable\_scope().reuse == True  
        with tf.variable\_scope("bar"):  
            # Now sub-scope inherits the reuse flag.  
            assert tf.get\_variable\_scope().reuse == True  
    # Exited the reusing scope, back to a non-reusing one.  
    assert tf.get\_variable\_scope().reuse == False

### Capturing variable scope

In all examples presented above, we shared parameters only because their names agreed, that is, because we opened a reusing variable scope with exactly the same string. In more complex cases, it might be useful to pass a VariableScope object rather than rely on getting the names right. To this end, variable scopes can be captured and used instead of names when opening a new variable scope.

with tf.variable\_scope("foo") as foo\_scope:  
    v = tf.get\_variable("v", [1])  
with tf.variable\_scope(foo\_scope):  
    w = tf.get\_variable("w", [1])  
with tf.variable\_scope(foo\_scope, reuse=True):  
    v1 = tf.get\_variable("v", [1])  
    w1 = tf.get\_variable("w", [1])  
assert v1 is v  
assert w1 is w

When opening a variable scope using a previously existing scope we jump out of the current variable scope prefix to an entirely different one. This is fully independent of where we do it.

with tf.variable\_scope("foo") as foo\_scope:  
    assert foo\_scope.name == "foo"  
with tf.variable\_scope("bar"):  
    with tf.variable\_scope("baz") as other\_scope:  
        assert other\_scope.name == "bar/baz"  
        with tf.variable\_scope(foo\_scope) as foo\_scope2:  
            assert foo\_scope2.name == "foo"  # Not changed.

### Initializers in variable scope

Using tf.get\_variable() allows to write functions that create or reuse variables and can be transparently called from outside. But what if we wanted to change the initializer of the created variables? Do we need to pass an extra argument to every function that creates variables? What about the most common case, when we want to set the default initializer for all variables in one place, on top of all functions? To help with these cases, variable scope can carry a default initializer. It is inherited by sub-scopes and passed to each tf.get\_variable() call. But it will be overridden if another initializer is specified explicitly.

with tf.variable\_scope("foo", initializer=tf.constant\_initializer(0.4)):  
    v = tf.get\_variable("v", [1])  
    assert v.eval() == 0.4  # Default initializer as set above.  
    w = tf.get\_variable("w", [1], initializer=tf.constant\_initializer(0.3)):  
    assert w.eval() == 0.3  # Specific initializer overrides the default.  
    with tf.variable\_scope("bar"):  
        v = tf.get\_variable("v", [1])  
        assert v.eval() == 0.4  # Inherited default initializer.  
    with tf.variable\_scope("baz", initializer=tf.constant\_initializer(0.2)):  
        v = tf.get\_variable("v", [1])  
        assert v.eval() == 0.2  # Changed default initializer.

### Names of ops in tf.variable\_scope()

We discussed how tf.variable\_scope governs the names of variables. But how does it influence the names of other ops in the scope? It is natural that ops created inside a variable scope should also share that name. For this reason, when we do with tf.variable\_scope("name"), this implicitly opens a tf.name\_scope("name"). For example:

with tf.variable\_scope("foo"):  
    x = 1.0 + tf.get\_variable("v", [1])  
assert x.op.name == "foo/add"

Name scopes can be opened in addition to a variable scope, and then they will only affect the names of the ops, but not of variables.

with tf.variable\_scope("foo"):  
    with tf.name\_scope("bar"):  
        v = tf.get\_variable("v", [1])  
        x = 1.0 + v  
assert v.name == "foo/v:0"  
assert x.op.name == "foo/bar/add"

When opening a variable scope using a captured object instead of a string, we do not alter the current name scope for ops.

## Examples of Use

Here are pointers to a few files that make use of variable scope. They can all be found in the [TensorFlow models repo](https://github.com/tensorflow/models). In particular, variable scope is heavily used for recurrent neural networks and sequence-to-sequence models.

| File | What's in it? |
| --- | --- |
| models/tutorials/image/cifar10/cifar10.py | Model for detecting objects in images. |
| models/tutorials/rnn/rnn\_cell.py | Cell functions for recurrent neural networks. |
| models/tutorials/rnn/seq2seq.py | Functions for building sequence-to-sequence models. |

# Threading and Queues

Queues are a powerful mechanism for asynchronous computation using TensorFlow.

Like everything in TensorFlow, a queue is a node in a TensorFlow graph. It's a stateful node, like a variable: other nodes can modify its content. In particular, nodes can enqueue new items in to the queue, or dequeue existing items from the queue.

To get a feel for queues, let's consider a simple example. We will create a "first in, first out" queue (FIFOQueue) and fill it with zeros. Then we'll construct a graph that takes an item off the queue, adds one to that item, and puts it back on the end of the queue. Slowly, the numbers on the queue increase.

Enqueue, EnqueueMany, and Dequeue are special nodes. They take a pointer to the queue instead of a normal value, allowing them to change it. We recommend you think of these as being like methods of the queue. In fact, in the Python API, they are methods of the queue object (e.g. q.enqueue(...)).

**N.B.** Queue methods (such as q.enqueue(...)) must run on the same device as the queue. Incompatible device placement directives will be ignored when creating these operations.

Now that you have a bit of a feel for queues, let's dive into the details...

## Queue usage overview

Queues, such as [tf.FIFOQueue](https://www.tensorflow.org/api_docs/python/tf/FIFOQueue) and [tf.RandomShuffleQueue](https://www.tensorflow.org/api_docs/python/tf/RandomShuffleQueue), are important TensorFlow objects for computing tensors asynchronously in a graph.

For example, a typical input architecture is to use a RandomShuffleQueue to prepare inputs for training a model:

* Multiple threads prepare training examples and push them in the queue.
* A training thread executes a training op that dequeues mini-batches from the queue

This architecture has many benefits, as highlighted in the [Reading data how to](https://www.tensorflow.org/programmers_guide/reading_data), which also gives an overview of functions that simplify the construction of input pipelines.

The TensorFlow Session object is multithreaded, so multiple threads can easily use the same session and run ops in parallel. However, it is not always easy to implement a Python program that drives threads as described above. All threads must be able to stop together, exceptions must be caught and reported, and queues must be properly closed when stopping.

TensorFlow provides two classes to help: [tf.train.Coordinator](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) and [tf.train.QueueRunner](https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner). These two classes are designed to be used together. The Coordinator class helps multiple threads stop together and report exceptions to a program that waits for them to stop. The QueueRunnerclass is used to create a number of threads cooperating to enqueue tensors in the same queue.

## Coordinator

The Coordinator class helps multiple threads stop together.

Its key methods are:

* [tf.train.Coordinator.should\_stop](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#should_stop): returns True if the threads should stop.
* [tf.train.Coordinator.request\_stop](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#request_stop): requests that threads should stop.
* [tf.train.Coordinator.join](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#join): waits until the specified threads have stopped.

You first create a Coordinator object, and then create a number of threads that use the coordinator. The threads typically run loops that stop when should\_stop() returns True.

Any thread can decide that the computation should stop. It only has to call request\_stop() and the other threads will stop as should\_stop() will then return True.

# Thread body: loop until the coordinator indicates a stop was requested.  
# If some condition becomes true, ask the coordinator to stop.  
def MyLoop(coord):  
  while not coord.should\_stop():  
    ...do something...  
    if ...some condition...:  
      coord.request\_stop()  
  
# Main thread: create a coordinator.  
coord = tf.train.Coordinator()  
  
# Create 10 threads that run 'MyLoop()'  
threads = [threading.Thread(target=MyLoop, args=(coord,)) for i in xrange(10)]  
  
# Start the threads and wait for all of them to stop.  
for t in threads:  
  t.start()  
coord.join(threads)

Obviously, the coordinator can manage threads doing very different things. They don't have to be all the same as in the example above. The coordinator also has support to capture and report exceptions. See the [tf.train.Coordinator](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) documentation for more details.

## QueueRunner

The QueueRunner class creates a number of threads that repeatedly run an enqueue op. These threads can use a coordinator to stop together. In addition, a queue runner runs a closer thread that automatically closes the queue if an exception is reported to the coordinator.

You can use a queue runner to implement the architecture described above.

First build a graph that uses a TensorFlow queue (e.g. a tf.RandomShuffleQueue) for input examples. Add ops that process examples and enqueue them in the queue. Add training ops that start by dequeueing from the queue.

example = ...ops to create one example...  
# Create a queue, and an op that enqueues examples one at a time in the queue.  
queue = tf.RandomShuffleQueue(...)  
enqueue\_op = queue.enqueue(example)  
# Create a training graph that starts by dequeuing a batch of examples.  
inputs = queue.dequeue\_many(batch\_size)  
train\_op = ...use 'inputs' to build the training part of the graph...

In the Python training program, create a QueueRunner that will run a few threads to process and enqueue examples. Create a Coordinator and ask the queue runner to start its threads with the coordinator. Write a training loop that also uses the coordinator.

# Create a queue runner that will run 4 threads in parallel to enqueue  
# examples.  
qr = tf.train.QueueRunner(queue, [enqueue\_op] \* 4)  
  
# Launch the graph.  
sess = tf.Session()  
# Create a coordinator, launch the queue runner threads.  
coord = tf.train.Coordinator()  
enqueue\_threads = qr.create\_threads(sess, coord=coord, start=True)  
# Run the training loop, controlling termination with the coordinator.  
for step in xrange(1000000):  
    if coord.should\_stop():  
        break  
    sess.run(train\_op)  
# When done, ask the threads to stop.  
coord.request\_stop()  
# And wait for them to actually do it.  
coord.join(enqueue\_threads)

## Handling exceptions

Threads started by queue runners do more than just run the enqueue ops. They also catch and handle exceptions generated by queues, including the tf.errors.OutOfRangeError exception, which is used to report that a queue was closed.

A training program that uses a coordinator must similarly catch and report exceptions in its main loop.

Here is an improved version of the training loop above.

try:  
    for step in xrange(1000000):  
        if coord.should\_stop():  
            break  
        sess.run(train\_op)  
except Exception, e:  
    # Report exceptions to the coordinator.  
    coord.request\_stop(e)  
finally:  
    # Terminate as usual. It is safe to call `coord.request\_stop()` twice.  
    coord.request\_stop()  
    coord.join(threads)

# Reading data

There are three main methods of getting data into a TensorFlow program:

* Feeding: Python code provides the data when running each step.
* Reading from files: an input pipeline reads the data from files at the beginning of a TensorFlow graph.
* Preloaded data: a constant or variable in the TensorFlow graph holds all the data (for small data sets).

## Feeding

TensorFlow's feed mechanism lets you inject data into any Tensor in a computation graph. A python computation can thus feed data directly into the graph.

Supply feed data through the feed\_dict argument to a run() or eval() call that initiates computation.

with tf.Session():  
  input = tf.placeholder(tf.float32)  
  classifier = ...  
  print(classifier.eval(feed\_dict={input: my\_python\_preprocessing\_fn()}))

While you can replace any Tensor with feed data, including variables and constants, the best practice is to use a[tf.placeholder](https://www.tensorflow.org/api_docs/python/tf/placeholder) node. A placeholder exists solely to serve as the target of feeds. It is not initialized and contains no data. A placeholder generates an error if it is executed without a feed, so you won't forget to feed it.

An example using placeholder and feeding to train on MNIST data can be found in[tensorflow/examples/tutorials/mnist/fully\_connected\_feed.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/fully_connected_feed.py), and is described in the [MNIST tutorial](https://www.tensorflow.org/get_started/mnist/mechanics).

## Reading from files

A typical pipeline for reading records from files has the following stages:

1. The list of filenames
2. Optional filename shuffling
3. Optional epoch limit
4. Filename queue
5. A Reader for the file format
6. A decoder for a record read by the reader
7. Optional preprocessing
8. Example queue

### Filenames, shuffling, and epoch limits

For the list of filenames, use either a constant string Tensor (like ["file0", "file1"] or [("file%d" % i) for i in range(2)]) or the [tf.train.match\_filenames\_once](https://www.tensorflow.org/api_docs/python/tf/train/match_filenames_once) function.

Pass the list of filenames to the [tf.train.string\_input\_producer](https://www.tensorflow.org/api_docs/python/tf/train/string_input_producer) function. string\_input\_producer creates a FIFO queue for holding the filenames until the reader needs them.

string\_input\_producer has options for shuffling and setting a maximum number of epochs. A queue runner adds the whole list of filenames to the queue once for each epoch, shuffling the filenames within an epoch if shuffle=True. This procedure provides a uniform sampling of files, so that examples are not under- or over- sampled relative to each other.

The queue runner works in a thread separate from the reader that pulls filenames from the queue, so the shuffling and enqueuing process does not block the reader.

### File formats

Select the reader that matches your input file format and pass the filename queue to the reader's read method. The read method outputs a key identifying the file and record (useful for debugging if you have some weird records), and a scalar string value. Use one (or more) of the decoder and conversion ops to decode this string into the tensors that make up an example.

#### CSV files

To read text files in [comma-separated value (CSV) format](https://tools.ietf.org/html/rfc4180), use a [tf.TextLineReader](https://www.tensorflow.org/api_docs/python/tf/TextLineReader) with the [tf.decode\_csv](https://www.tensorflow.org/api_docs/python/tf/decode_csv)operation. For example:

filename\_queue = tf.train.string\_input\_producer(["file0.csv", "file1.csv"])  
  
reader = tf.TextLineReader()  
key, value = reader.read(filename\_queue)  
  
# Default values, in case of empty columns. Also specifies the type of the  
# decoded result.  
record\_defaults = [[1], [1], [1], [1], [1]]  
col1, col2, col3, col4, col5 = tf.decode\_csv(  
    value, record\_defaults=record\_defaults)  
features = tf.stack([col1, col2, col3, col4])  
  
with tf.Session() as sess:  
  # Start populating the filename queue.  
  coord = tf.train.Coordinator()  
  threads = tf.train.start\_queue\_runners(coord=coord)  
  
  for i in range(1200):  
    # Retrieve a single instance:  
    example, label = sess.run([features, col5])  
  
  coord.request\_stop()  
  coord.join(threads)

Each execution of read reads a single line from the file. The decode\_csv op then parses the result into a list of tensors. The record\_defaults argument determines the type of the resulting tensors and sets the default value to use if a value is missing in the input string.

You must call tf.train.start\_queue\_runners to populate the queue before you call run or eval to execute the read. Otherwise read will block while it waits for filenames from the queue.

#### Fixed length records

To read binary files in which each record is a fixed number of bytes, use [tf.FixedLengthRecordReader](https://www.tensorflow.org/api_docs/python/tf/FixedLengthRecordReader) with the [tf.decode\_raw](https://www.tensorflow.org/api_docs/python/tf/decode_raw) operation. The decode\_raw op converts from a string to a uint8 tensor.

For example, [the CIFAR-10 dataset](http://www.cs.toronto.edu/~kriz/cifar.html) uses a file format where each record is represented using a fixed number of bytes: 1 byte for the label followed by 3072 bytes of image data. Once you have a uint8 tensor, standard operations can slice out each piece and reformat as needed. For CIFAR-10, you can see how to do the reading and decoding in [tensorflow\_models/tutorials/image/cifar10/cifar10\_input.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow_models/tutorials/image/cifar10/cifar10_input.py) and described in [this tutorial](https://www.tensorflow.org/tutorials/deep_cnn#prepare_the_data).

#### Standard TensorFlow format

Another approach is to convert whatever data you have into a supported format. This approach makes it easier to mix and match data sets and network architectures. The recommended format for TensorFlow is a [TFRecords file](https://www.tensorflow.org/api_guides/python/python_io#tfrecords_format_details)containing [tf.train.Example protocol buffers](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/example/example.proto) (which contain [Features](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/example/feature.proto) as a field). You write a little program that gets your data, stuffs it in an Example protocol buffer, serializes the protocol buffer to a string, and then writes the string to a TFRecords file using the [tf.python\_io.TFRecordWriter](https://www.tensorflow.org/api_docs/python/tf/python_io/TFRecordWriter). For example,[tensorflow/examples/how\_tos/reading\_data/convert\_to\_records.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/how_tos/reading_data/convert_to_records.py) converts MNIST data to this format.

To read a file of TFRecords, use [tf.TFRecordReader](https://www.tensorflow.org/api_docs/python/tf/TFRecordReader) with the [tf.parse\_single\_example](https://www.tensorflow.org/api_docs/python/tf/parse_single_example) decoder. The parse\_single\_example op decodes the example protocol buffers into tensors. An MNIST example using the data produced by convert\_to\_records can be found in[tensorflow/examples/how\_tos/reading\_data/fully\_connected\_reader.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/how_tos/reading_data/fully_connected_reader.py), which you can compare with the fully\_connected\_feed version.

### Preprocessing

You can then do any preprocessing of these examples you want. This would be any processing that doesn't depend on trainable parameters. Examples include normalization of your data, picking a random slice, adding noise or distortions, etc. See [tensorflow\_models/tutorials/image/cifar10/cifar10\_input.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow_models/tutorials/image/cifar10/cifar10_input.py) for an example.

### Batching

At the end of the pipeline we use another queue to batch together examples for training, evaluation, or inference. For this we use a queue that randomizes the order of examples, using the [tf.train.shuffle\_batch](https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch).

Example:

def read\_my\_file\_format(filename\_queue):  
  reader = tf.SomeReader()  
  key, record\_string = reader.read(filename\_queue)  
  example, label = tf.some\_decoder(record\_string)  
  processed\_example = some\_processing(example)  
  return processed\_example, label  
  
def input\_pipeline(filenames, batch\_size, num\_epochs=None):  
  filename\_queue = tf.train.string\_input\_producer(  
      filenames, num\_epochs=num\_epochs, shuffle=True)  
  example, label = read\_my\_file\_format(filename\_queue)  
  # min\_after\_dequeue defines how big a buffer we will randomly sample  
  #   from -- bigger means better shuffling but slower start up and more  
  #   memory used.  
  # capacity must be larger than min\_after\_dequeue and the amount larger  
  #   determines the maximum we will prefetch.  Recommendation:  
  #   min\_after\_dequeue + (num\_threads + a small safety margin) \* batch\_size  
  min\_after\_dequeue = 10000  
  capacity = min\_after\_dequeue + 3 \* batch\_size  
  example\_batch, label\_batch = tf.train.shuffle\_batch(  
      [example, label], batch\_size=batch\_size, capacity=capacity,  
      min\_after\_dequeue=min\_after\_dequeue)  
  return example\_batch, label\_batch

If you need more parallelism or shuffling of examples between files, use multiple reader instances using the[tf.train.shuffle\_batch\_join](https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch_join). For example:

def read\_my\_file\_format(filename\_queue):  
  # Same as above  
  
def input\_pipeline(filenames, batch\_size, read\_threads, num\_epochs=None):  
  filename\_queue = tf.train.string\_input\_producer(  
      filenames, num\_epochs=num\_epochs, shuffle=True)  
  example\_list = [read\_my\_file\_format(filename\_queue)  
                  for \_ in range(read\_threads)]  
  min\_after\_dequeue = 10000  
  capacity = min\_after\_dequeue + 3 \* batch\_size  
  example\_batch, label\_batch = tf.train.shuffle\_batch\_join(  
      example\_list, batch\_size=batch\_size, capacity=capacity,  
      min\_after\_dequeue=min\_after\_dequeue)  
  return example\_batch, label\_batch

You still only use a single filename queue that is shared by all the readers. That way we ensure that the different readers use different files from the same epoch until all the files from the epoch have been started. (It is also usually sufficient to have a single thread filling the filename queue.)

An alternative is to use a single reader via the [tf.train.shuffle\_batch](https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch) with num\_threads bigger than 1. This will make it read from a single file at the same time (but faster than with 1 thread), instead of N files at once. This can be important:

* If you have more reading threads than input files, to avoid the risk that you will have two threads reading the same example from the same file near each other.
* Or if reading N files in parallel causes too many disk seeks.

How many threads do you need? the tf.train.shuffle\_batch\* functions add a summary to the graph that indicates how full the example queue is. If you have enough reading threads, that summary will stay above zero. You can [view your summaries as training progresses using TensorBoard](https://www.tensorflow.org/get_started/summaries_and_tensorboard).

### Creating threads to prefetch using QueueRunner objects

The short version: many of the tf.train functions listed above add [tf.train.QueueRunner](https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) objects to your graph. These require that you call [tf.train.start\_queue\_runners](https://www.tensorflow.org/api_docs/python/tf/train/start_queue_runners) before running any training or inference steps, or it will hang forever. This will start threads that run the input pipeline, filling the example queue so that the dequeue to get the examples will succeed. This is best combined with a [tf.train.Coordinator](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) to cleanly shut down these threads when there are errors. If you set a limit on the number of epochs, that will use an epoch counter that will need to be initialized. The recommended code pattern combining these is:

# Create the graph, etc.  
init\_op = tf.global\_variables\_initializer()  
  
# Create a session for running operations in the Graph.  
sess = tf.Session()  
  
# Initialize the variables (like the epoch counter).  
sess.run(init\_op)  
  
# Start input enqueue threads.  
coord = tf.train.Coordinator()  
threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)  
  
try:  
    while not coord.should\_stop():  
        # Run training steps or whatever  
        sess.run(train\_op)  
  
except tf.errors.OutOfRangeError:  
    print('Done training -- epoch limit reached')  
finally:  
    # When done, ask the threads to stop.  
    coord.request\_stop()  
  
# Wait for threads to finish.  
coord.join(threads)  
sess.close()

#### Aside: What is happening here?

First we create the graph. It will have a few pipeline stages that are connected by queues. The first stage will generate filenames to read and enqueue them in the filename queue. The second stage consumes filenames (using a Reader), produces examples, and enqueues them in an example queue. Depending on how you have set things up, you may actually have a few independent copies of the second stage, so that you can read from multiple files in parallel. At the end of these stages is an enqueue operation, which enqueues into a queue that the next stage dequeues from. We want to start threads running these enqueuing operations, so that our training loop can dequeue examples from the example queue.

The helpers in tf.train that create these queues and enqueuing operations add a [tf.train.QueueRunner](https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) to the graph using the [tf.train.add\_queue\_runner](https://www.tensorflow.org/api_docs/python/tf/train/add_queue_runner) function. Each QueueRunner is responsible for one stage, and holds the list of enqueue operations that need to be run in threads. Once the graph is constructed, the[tf.train.start\_queue\_runners](https://www.tensorflow.org/api_docs/python/tf/train/start_queue_runners) function asks each QueueRunner in the graph to start its threads running the enqueuing operations.

If all goes well, you can now run your training steps and the queues will be filled by the background threads. If you have set an epoch limit, at some point an attempt to dequeue examples will get an[tf.errors.OutOfRangeError](https://www.tensorflow.org/api_docs/python/tf/errors/OutOfRangeError). This is the TensorFlow equivalent of "end of file" (EOF) -- this means the epoch limit has been reached and no more examples are available.

The last ingredient is the [tf.train.Coordinator](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator). This is responsible for letting all the threads know if anything has signalled a shut down. Most commonly this would be because an exception was raised, for example one of the threads got an error when running some operation (or an ordinary Python exception).

For more about threading, queues, QueueRunners, and Coordinators [see here](https://www.tensorflow.org/programmers_guide/threading_and_queues).

#### Aside: How clean shut-down when limiting epochs works

Imagine you have a model that has set a limit on the number of epochs to train on. That means that the thread generating filenames will only run that many times before generating an OutOfRange error. The QueueRunner will catch that error, close the filename queue, and exit the thread. Closing the queue does two things:

* Any future attempt to enqueue in the filename queue will generate an error. At this point there shouldn't be any threads trying to do that, but this is helpful when queues are closed due to other errors.
* Any current or future dequeue will either succeed (if there are enough elements left) or fail (with an OutOfRange error) immediately. They won't block waiting for more elements to be enqueued, since by the previous point that can't happen.

The point is that when the filename queue is closed, there will likely still be many filenames in that queue, so the next stage of the pipeline (with the reader and other preprocessing) may continue running for some time. Once the filename queue is exhausted, though, the next attempt to dequeue a filename (e.g. from a reader that has finished with the file it was working on) will trigger an OutOfRange error. In this case, though, you might have multiple threads associated with a single QueueRunner. If this isn't the last thread in the QueueRunner, the OutOfRange error just causes the one thread to exit. This allows the other threads, which are still finishing up their last file, to proceed until they finish as well. (Assuming you are using a [tf.train.Coordinator](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator), other types of errors will cause all the threads to stop.) Once all the reader threads hit the OutOfRange error, only then does the next queue, the example queue, gets closed.

Again, the example queue will have some elements queued, so training will continue until those are exhausted. If the example queue is a [tf.RandomShuffleQueue](https://www.tensorflow.org/api_docs/python/tf/RandomShuffleQueue), say because you are using shuffle\_batch or shuffle\_batch\_join, it normally will avoid ever having fewer than its min\_after\_dequeue attr elements buffered. However, once the queue is closed that restriction will be lifted and the queue will eventually empty. At that point the actual training threads, when they try and dequeue from example queue, will start getting OutOfRange errors and exiting. Once all the training threads are done, [tf.train.Coordinator.join](https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#join) will return and you can exit cleanly.

### Filtering records or producing multiple examples per record

Instead of examples with shapes [x, y, z], you will produce a batch of examples with shape [batch, x, y, z]. The batch size can be 0 if you want to filter this record out (maybe it is in a hold-out set?), or bigger than 1 if you are producing multiple examples per record. Then simply set enqueue\_many=True when calling one of the batching functions (such as shuffle\_batch or shuffle\_batch\_join).

### Sparse input data

SparseTensors don't play well with queues. If you use SparseTensors you have to decode the string records using[tf.parse\_example](https://www.tensorflow.org/api_docs/python/tf/parse_example) **after** batching (instead of using tf.parse\_single\_example before batching).

## Preloaded data

This is only used for small data sets that can be loaded entirely in memory. There are two approaches:

* Store the data in a constant.
* Store the data in a variable, that you initialize and then never change.

Using a constant is a bit simpler, but uses more memory (since the constant is stored inline in the graph data structure, which may be duplicated a few times).

training\_data = ...  
training\_labels = ...  
with tf.Session():  
  input\_data = tf.constant(training\_data)  
  input\_labels = tf.constant(training\_labels)  
  ...

To instead use a variable, you need to also initialize it after the graph has been built.

training\_data = ...  
training\_labels = ...  
with tf.Session() as sess:  
  data\_initializer = tf.placeholder(dtype=training\_data.dtype,  
                                    shape=training\_data.shape)  
  label\_initializer = tf.placeholder(dtype=training\_labels.dtype,  
                                     shape=training\_labels.shape)  
  input\_data = tf.Variable(data\_initializer, trainable=False, collections=[])  
  input\_labels = tf.Variable(label\_initializer, trainable=False, collections=[])  
  ...  
  sess.run(input\_data.initializer,  
           feed\_dict={data\_initializer: training\_data})  
  sess.run(input\_labels.initializer,  
           feed\_dict={label\_initializer: training\_labels})

Setting trainable=False keeps the variable out of the GraphKeys.TRAINABLE\_VARIABLES collection in the graph, so we won't try and update it when training. Setting collections=[] keeps the variable out of theGraphKeys.GLOBAL\_VARIABLES collection used for saving and restoring checkpoints.

Either way, [tf.train.slice\_input\_producer](https://www.tensorflow.org/api_docs/python/tf/train/slice_input_producer) can be used to produce a slice at a time. This shuffles the examples across an entire epoch, so further shuffling when batching is undesirable. So instead of using the shuffle\_batch functions, we use the plain [tf.train.batch](https://www.tensorflow.org/api_docs/python/tf/train/batch) function. To use multiple preprocessing threads, set the num\_threads parameter to a number bigger than 1.

An MNIST example that preloads the data using constants can be found in[tensorflow/examples/how\_tos/reading\_data/fully\_connected\_preloaded.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/how_tos/reading_data/fully_connected_preloaded.py), and one that preloads the data using variables can be found in[tensorflow/examples/how\_tos/reading\_data/fully\_connected\_preloaded\_var.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/how_tos/reading_data/fully_connected_preloaded_var.py), You can compare these with the fully\_connected\_feed and fully\_connected\_reader versions above.

## Multiple input pipelines

Commonly you will want to train on one dataset and evaluate (or "eval") on another. One way to do this is to actually have two separate processes:

* The training process reads training input data and periodically writes checkpoint files with all the trained variables.
* The evaluation process restores the checkpoint files into an inference model that reads validation input data.

This is what is done in [the example CIFAR-10 model](https://www.tensorflow.org/tutorials/deep_cnn#save_and_restore_checkpoints). This has a couple of benefits:

* The eval is performed on a single snapshot of the trained variables.
* You can perform the eval even after training has completed and exited.

You can have the train and eval in the same graph in the same process, and share their trained variables. See [the shared variables tutorial](https://www.tensorflow.org/programmers_guide/variable_scope).

# Supervisor: Training Helper for Days-Long Trainings.

To train a model with TensorFlow you can simply run a training op a number of times and save a checkpoint of the trained parameters when you're done. This works well for small models that can train in a few hours.

Larger models that require days of training, possibly across multiple replicas, need a more robust training process that:

* Handles shutdowns and crashes cleanly.
* Can be resumed after a shutdown or a crash.
* Can be monitored through TensorBoard.

To be able to resume training after a shutdown or a crash the training process must save checkpoints regularly. On restart, it must look for the most recent checkpoint and load it before resuming training.

To be monitored through TensorBoard, the training process must run summary ops regularly and append the returned values to an events file as explained in [TensorBoard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard). TensorBoard monitors events files and displays graphs reporting training progress over time.

The [tf.train.Supervisor](https://www.tensorflow.org/api_docs/python/tf/train/Supervisor) provides a set of services that helps implement a robust training process.

This how-to shows how to use the supervisor directly. Please also consider using one of several frameworks built on top of the supervisor that provide richer training loops, and numerous customization options: [tf.learn](https://www.tensorflow.org/api_guides/python/contrib.learn) is a good choice.

Note that the supervisor is very helpful for training large models, but can also be used for smaller models without any penalty.

## Very Simple Scenario

The simplest scenario for using a supervisor is to:

* Create a Supervisor object, passing it the path to a directory where to save checkpoints and summaries.
* Ask the supervisor for a session with [tf.train.Supervisor.managed\_session](https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#managed_session).
* Use the session to execute a train op, checking at each step if the supervisor requests that the training stops.

  ...create graph...  
  my\_train\_op = ...  
  
  sv = tf.train.Supervisor(logdir="/my/training/directory")  
  with sv.managed\_session() as sess:  
    for step in range(100000):  
      if sv.should\_stop():  
        break  
      sess.run(my\_train\_op)

### Started Services

In the very simple scenario, the managed\_session() call starts a few services, which run in their own threads, and use the managed session to run ops in your graph.

If your graph contains an integer variable named global\_step, the services use its value to measure the number of training steps executed. See the [MNIST training tutorial](https://www.tensorflow.org/get_started/mnist/mechanics#training) for how to create a global\_step variable.

* Checkpointing service: Saves a copy of the graph variables in the logdir. The checkpoint filename uses the value of the global\_step variable if one was added to your graph. Runs every 10 minutes by default.
* Summary service: Runs all the summary ops and appends their output to an [events file](https://www.tensorflow.org/get_started/summaries_and_tensorboard) in the logdir. Runs every 2 minutes by default.
* Step counter: Counts how many steps have been executed, by looking at changes in the global\_step variable. Appends a summary to the events file reporting the number of global steps per second. The summary tag is "global\_step/sec". This also runs every 2 minutes by default.
* Queue Runners: If any [tf.train.QueueRunner](https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) were added to the graph, the supervisor launches them in their own threads.

All time intervals can be changed when constructing the supervisor object. See the [supervisor reference](https://www.tensorflow.org/programmers_guide/supervisor#supervisor_reference) for details.

### Checking for Stop

The check for stop in the main training loop is important and necessary.

Exceptions raised in the service threads are reported to the supervisor which then sets its should\_stop() condition to true. Other service threads notice that condition and terminate properly. The main training loop, within the managed\_session() block, must also check for the stop condition and terminate.

Note that managed\_session() takes care of catching exceptions raised from the training loop to report them to the supervisor. The main loop does not need to do anything special about exceptions. It only needs to check for the stop condition.

### Recovery

If the training program shuts down or crashes, its most recent checkpoint and event files are left in the logdir. When you restart the program, managed\_session() restores the graph from the most recent checkpoint and resumes training where it stopped.

A new events file is created. If you start TensorBoard and point it to the logdir, it will know how to merge the contents of the two events files and will show the training resuming at the last global step from the checkpoint.

## Larger Model Scenario

The very simple scenario is sufficient for most small to medium sized models. Larger models may run out memory when the summary service runs: The summary ops are run in parallel with the main loop running the train op. This can cause memory usage to peak to up to two times the normal use.

For a larger model you can tell the supervisor to not run the summary service and instead run it yourself in your main training loop: pass summary\_op=None when constructing the supervisor.

For example this code runs the summary op every 100 steps in the training loop:

  ...create graph...  
  my\_train\_op = ...  
  my\_summary\_op = tf.summary.merge\_all()  
  
  sv = tf.train.Supervisor(logdir="/my/training/directory",  
                     summary\_op=None) # Do not run the summary service  
  with sv.managed\_session() as sess:  
    for step in range(100000):  
      if sv.should\_stop():  
        break  
      if step % 100 == 0:  
        \_, summ = session.run([my\_train\_op, my\_summary\_op])  
        sv.summary\_computed(sess, summ)  
      else:  
        session.run(my\_train\_op)

## Pre-trained Model Scenario

The managed\_session() call takes care of initializing the model in the session. The model is restored from a checkpoint if one is available, or initialized from scratch otherwise.

One common scenario is to initialize the model by loading a "pre-trained" checkpoint that was saved while training a usually slightly different model using a different dataset.

You can load a pre-trained checkpoint by passing an "init function" to the supervisor. This function is called only if the model needs to be initialized from scratch, not when the model can be recovered from a checkpoint from the logdir.

To load the pre-trained model, the init function needs a [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver) object, so you should create a saver for this purpose. This is usually a good idea because the new model may contain variables that are not present in the pre-trained checkpoint: This saver must only restore the pre-trained variables. If you were using the default saver, you could get an error trying to restore all the variables of the new model from the pre-trained checkpoint.

  ...create graph...  
  # Create a saver that restores only the pre-trained variables.  
  pre\_train\_saver = tf.train.Saver([pre\_train\_var1, pre\_train\_var2])  
  
  # Define an init function that loads the pretrained checkpoint.  
  def load\_pretrain(sess):  
    pre\_train\_saver.restore(sess, "<path to pre-trained-checkpoint>")  
  
  # Pass the init function to the supervisor.  
  #  
  # The init function is called \_after\_ the variables have been initialized  
  # by running the init\_op.  
  sv = tf.train.Supervisor(logdir="/my/training/directory",  
                     init\_fn=load\_pretrain)  
  with sv.managed\_session() as sess:  
    # Here sess was either initialized from the pre-trained-checkpoint or  
    # recovered from a checkpoint saved in a previous run of this code.  
    ...

## Running Your Own Services

Supervisor services, such as the checkpointing service, run in threads parallel to the main training loop. You sometimes want to add your own services, for example to fetch different sets of summaries on a different schedule than the usual summary service.

Use the [tf.train.Supervisor.loop](https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#loop) method of the supervisor for this purpose. It repeatedly calls a function of your choice on a timer until the supervisor stop condition becomes true, so it plays nicely with the other services.

Example: Call my\_additional\_summaries() every 20mn:

def my\_additional\_sumaries(sv, sess):  
 ...fetch and write summaries, see below...  
  
...  
  sv = tf.train.Supervisor(logdir="/my/training/directory")  
  with sv.managed\_session() as sess:  
    # Call my\_additional\_sumaries() every 1200s, or 20mn,  
    # passing (sv, sess) as arguments.  
    sv.loop(1200, my\_additional\_sumaries, args=(sv, sess))  
    ...main training loop...

## Writing Summaries

The supervisor always creates an events file in its logdir, as well as a [tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter)to append events and summaries to that file. If you want to write your own summaries it is a good idea to append them to that same events file: TensorBoard likes it better when only one events file in a directory is being actively appended to.

The supervisor provides a helper function to append summaries:[tf.train.Supervisor.summary\_computed](https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#summary_computed). Just pass to the function the output returned by a summary op. Here is an example of using that function to implement my\_additional\_sumaries() from the previous example:

def my\_additional\_sumaries(sv, sess):  
  summaries = sess.run(my\_additional\_summary\_op)  
  sv.summary\_computed(sess, summaries)

For more advanced usages, the supervisor provides access to its summary writer through its[tf.train.Supervisor.summary\_writer](https://www.tensorflow.org/api_docs/python/tf/train/Supervisor#summary_writer) attribute.

## Supervisor Reference

The [Very Simple Scenario](https://www.tensorflow.org/programmers_guide/supervisor#very_simple_scenario), and the [Larger Model Scenario](https://www.tensorflow.org/programmers_guide/supervisor#larger_model_scenario) show basic uses of a supervisor. More advanced scenarios can be constructed by using the many options provided by the supervisor

### Checkpointing: Where and When.

The managed\_session() call launches the checkpointing service, which can be configured by the following keyword arguments to the Supervisor() constructor:

* logdir: path to a directory where the checkpointing service creates checkpoints. The directory is created if needed. Passing None disables the checkpointing and the summary services.
* checkpoint\_basename: Name of the checkpoint files to create, defaults to "model.ckpt".

If the model contains a scalar integer variable named global\_step, the value of that variable is appended to the checkpoint filename.

For example, at global step 1234 the checkpoint filename is "model.ckpt-1234".

* save\_model\_secs: Number of seconds between each checkpoint. Defaults to 600, or 10 minutes.

When choosing a value, consider how much work you want to lose in case of a crash: you will never lose more than save\_model\_secs seconds of work. Setting this to 0 disables the checkpointing service.

* saver: A [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver) object to use for checkpointing.

If you do not pass one, the supervisor creates one for you by calling tf.train.Saver(), which add ops to save and restore all variables in your model. This is usually what you need.

Example: Use a custom Saver and checkpoint every 30 seconds.

  ...create graph...  
  my\_saver = tf.train.Saver(<only some variables>)  
  sv = tf.train.Supervisor(logdir="/my/training/directory",  
                     saver=my\_saver,  
                     save\_model\_secs=30)  
  with sv.managed\_session() as sess:  
    ...training loop...

### Summaries: Where and When.

The managed\_session() call launches the summary service which fetches summaries and reports the number of steps executed per second. It can be configured by the following keyword arguments to the Supervisor() constructor:

* logdir: Path to a directory where the summary service creates event files. The directory is created if needed. Passing None disables the summary service as well as the checkpointing services.
* save\_summaries\_secs: Number of seconds between each run of the summary service. Defaults to 120, or 2 minutes.

When choosing a value, consider how expensive your summaries are, and how much disk they will occupy. Pass 0 to disable the summary service.

* summary\_op: Op to use to fetch the summaries.

If not specified, the supervisor use the first op in the tf.GraphKeys.SUMMARY\_OP [graph collection](https://www.tensorflow.org/api_docs/python/tf/Graph#add_to_collection). If the collection is empty the supervisor creates an op that aggregates all summaries in the graph using tf.summary.merge\_all().

Passing None disables the summary service.

* global\_step: Tensor to use to count the global step.

If not specified, the supervisor uses the first tensor in the tf.GraphKeys.GLOBAL\_STEP [graph collection](https://www.tensorflow.org/api_docs/python/tf/Graph#add_to_collection). If the collection is empty, the supervisor looks for a scalar integer variable named global\_step in the graph.

If found, the global step tensor is used to measure the number of training steps executed. Note that your training op is responsible for incrementing the global step value.

### Model Initialization and Recovery

The managed\_session() call takes care of initializing or recovering a session. It returns a session with a fully initialized model, ready to run ops. If a checkpoint exists in the logdir when managed\_session() is called, the model is initialized by loading that checkpoint, otherwise it is initialized by calling an init op and optionally an init function.

When no checkpoint is available, model initialization is controlled by the following keyword arguments to the Supervisor() constructor:

* init\_op: Op to run to initialize the model.

If not specified, the supervisor uses the first op in the tf.GraphKeys.INIT\_OP collection. If the collection is empty, the supervisor adds an op to initialize all the variables in the graph by calling tf.global\_variables\_initializer().

Pass None to not use an init op.

* init\_fn: Python function to call to initialize the model.

If specified, called as init\_fn(sess) where sess is the managed session. If an init op is also used, the init function is called after the init op.

* local\_init\_op: An additional op to initialize parts of the graph that are not saved in checkpoints such as tables and [local variables](https://www.tensorflow.org/api_docs/python/tf/contrib/framework/local_variable). The local init op is run before the init op and the init function.

If not specified, the supervisor uses the first op in the tf.GraphKeys.LOCAL\_INIT\_OPcollection. If the collection is empty the supervisor adds an op to initialize all the tables and local variables in the graph by calling tf.initialize\_all\_tables() andtf.initialize\_all\_local\_variables().

Pass None to not use a local init op.

* ready\_op: Op to check if the model is initialized.

After running the local init op, the init op, and the init function, the supervisor verifies that the model is fully initialized by running the ready op. This is an op that returns an empty string if the model is initialized, or a description of what parts of the model are not initialized if not.

If not specified, the supervisor uses the first op in the tf.GraphKeys.READY\_OP collection. If the collection is empty the supervisor creates a ready op that verifies that all variables are initialized by calling tf.report\_uninitialized\_variables().

Pass None to disable the ready op. In that case the model is not checked after initialization.

Checkpoint recovery is controlled by the following keyword arguments to the Supervisor()constructor:

* logdir: Path to a directory in which to look for checkpoints. The checkpoint service saves a metadata file, named "checkpoint", in the checkpoint directory that indicates the path to the most recent checkpoint.

This file is in text format. When in a pinch, you can edit it manually to recover from a different checkpoint than the most recent one.

* ready\_op: (see above). The ready op is run before and after loading the checkpoint. The first run checks if the model needs to be initialized and the second run verifies that the model is fully initialized.
* local\_init\_op: (see above). The local init op is run before running the ready op the first time, to initialize local variables and tables.
* saver: (see above). Saver object used to load the checkpoint.

# TensorFlow Debugger (tfdbg) Command-Line-Interface Tutorial: MNIST

**(Experimental)**

TensorFlow debugger (**tfdbg**) is a specialized debugger for TensorFlow. It provides visibility into the internal structure and states of running TensorFlow graphs. The insight gained from this visibility should facilitate debugging of various types of model bugs during training and inference.

This tutorial showcases the features of tfdbg command-line interface (CLI), by focusing on how to debug a type of frequently-encountered bug in TensorFlow model development: bad numerical values (nans and infs) causing training to fail.

To **observe** such an issue, run the following code without the debugger:

python -m tensorflow.python.debug.examples.debug\_mnist

This code trains a simple NN for MNIST digit image recognition. Notice that the accuracy increases slightly after the first training step, but then gets stuck at a low (near-chance) level:



Scratching your head, you suspect that certain nodes in the training graph generated bad numeric values such as infs and nans. The computation-graph paradigm of TensorFlow makes it non-trivial to debug such model-internal states with general-purpose debuggers such as Python's pdb. **tfdbg** specializes in diagnosing these types of issues and pinpointing the exact node where the problem first surfaced.

## Wrapping TensorFlow Sessions with tfdbg

To add support for **tfdbg** in our example, we just need to add the following three lines of code, which wrap the Session object with a debugger wrapper when the --debug flag is provided:

# Let your BUILD target depend on "//tensorflow/python/debug:debug\_py"  
# (You don't need to worry about the BUILD dependency if you are using a pip  
#  install of open-source TensorFlow.)  
from tensorflow.python import debug as tf\_debug  
  
sess = tf\_debug.LocalCLIDebugWrapperSession(sess)  
sess.add\_tensor\_filter("has\_inf\_or\_nan", tf\_debug.has\_inf\_or\_nan)

This wrapper has the same interface as Session, so enabling debugging requires no other changes to the code. But the wrapper provides additional features including:

* Bringing up a terminal-based user interface (UI) before and after each run() call, to let you control the execution and inspect the graph's internal state.
* Allowing you to register special "filters" for tensor values, to facilitate the diagnosis of issues.

In this example, we are registering a tensor filter called [tfdbg.has\_inf\_or\_nan](https://www.tensorflow.org/api_docs/python/tfdbg/has_inf_or_nan), which simply determines if there are any nan or inf values in any intermediate tensor of the graph. (This filter is a common enough use case that we ship it with the [debug\_data](https://www.tensorflow.org/api_guides/python/tfdbg#Classes_for_debug_dump_data_and_directories) module.)

def has\_inf\_or\_nan(datum, tensor):  
  return np.any(np.isnan(tensor)) or np.any(np.isinf(tensor))

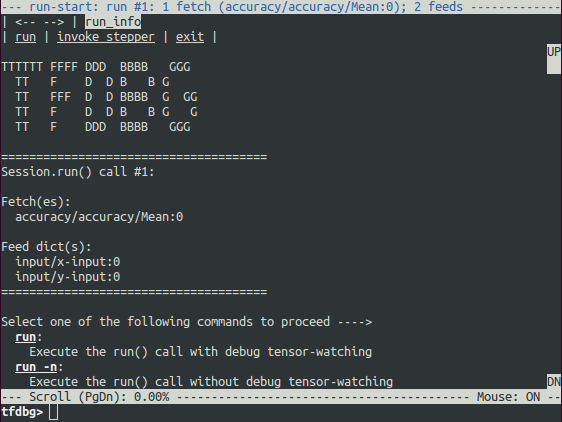
TIP: You can also write your own custom filters. See the [API documentation](https://www.tensorflow.org/api_docs/python/tfdbg/DebugDumpDir#find) of DebugDumpDir.find() for additional information.

## Debugging Model Training with tfdbg

Let's try training the model again with debugging enabled. Execute the command from above, this time with the --debug flag added:

python -m tensorflow.python.debug.examples.debug\_mnist --debug

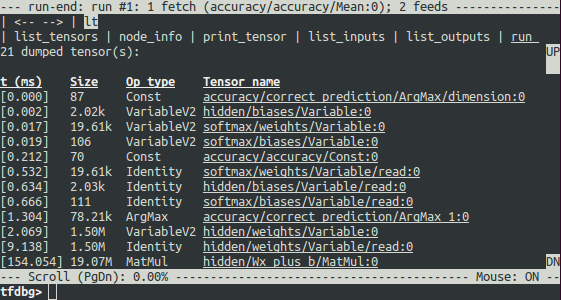
The debug wrapper session will prompt you when it is about to execute the first run() call, with information regarding the fetched tensor and feed dictionaries displayed on the screen.



This is what we refer to as the run-start UI. If the screen size is too small to display the content of the message in its entirety, you can resize it or use the **PageUp** / **PageDown** / **Home** / **End** keys to navigate the screen output.

As the screen output indicates, the first run() call calculates the accuracy using a test data set—i.e., a forward pass on the graph. You can enter the command run (or its shorthand r) to launch the run() call. On terminals that support mouse events, you can simply click the underlined run on the top left corner of the screen to proceed.

This will bring up another screen right after the run() call has ended, which will display all dumped intermediate tensors from the run. (These tensors can also be obtained by running the command lt after you executed run.) This is called the **run-end UI**:



### tfdbg CLI Frequently-Used Commands

Try the following commands at the tfdbg> prompt (referencing the code attensorflow/python/debug/examples/debug\_mnist.py):

| Command Example | Explanation |
| --- | --- |
| pt hidden/Relu:0 | Print the value of the tensor hidden/Relu:0. |
| pt hidden/Relu:0[0:50,:] | Print a subarray of the tensor hidden/Relu:0, using [numpy](http://www.numpy.org/)-style array slicing. |
| pt hidden/Relu:0[0:50,:] -a | For a large tensor like the one here, print its value in its entirety—i.e., without using any ellipsis. May take a long time for large tensors. |
| pt hidden/Relu:0[0:10,:] -a -r [1,inf] | Use the -r flag to highlight elements falling into the specified numerical range. Multiple ranges can be used in conjunction, e.g., -r [[-inf,-1],[1,inf]]. |
| @[10,0] or @10,0 | Navigate to indices [10, 0] in the tensor being displayed. |
| /inf | Search the screen output with the regex inf and highlight any matches. |
| / | Scroll to the next line with matches to the searched regex (if any). |
| ni -a hidden/Relu | Display information about the node hidden/Relu, including node attributes. |
| ni -t hidden/Relu | Display the stack trace of node hidden/Relu's construction. |
| li -r hidden/Relu:0 | List the inputs to the node hidden/Relu, recursively—i.e., the input tree. |
| lo -r hidden/Relu:0 | List the recipients of the output of the node hidden/Relu, recursively—i.e., the output recipient tree. |
| lt -n softmax.\* | List all dumped tensors whose names match the regular-expression pattern softmax.\*. |
| lt -t MatMul | List all dumped tensors whose node type is MatMul. |
| ps /path/to/source.py | Print the Python source file source.py, with the lines annotated with the ops created at each of them, respectively. |
| ps -t /path/to/source.py | Same as the command above, but perform annotation using dumped Tensors, instead of ops. |
| ps -b 30 /path/to/source.py | Annotate source.py beginning at line 30. |
| run\_info or ri | Display information about the current run, including fetches and feeds. |
| help | Print general help information listing all available **tfdbg** commands and their flags. |
| help lt | Print the help information for the lt command. |

In this first run() call, there happen to be no problematic numerical values. You can move on to the next run by using the command run or its shorthand r.

TIP: If you enter run or r repeatedly, you will be able to move through the run() calls in a sequential manner.

You can also use the -t flag to move ahead a number of run() calls at a time, for example:

tfdbg> run -t 10

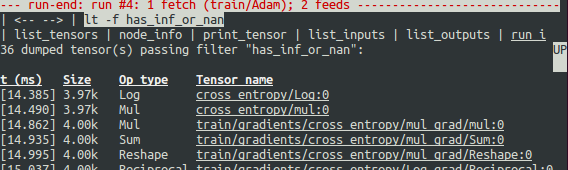
Instead of entering run repeatedly and manually searching for nans and infs in the run-end UI after every run() call, you can use the following command to let the debugger repeatedly execute run() calls without stopping at the run-start or run-end prompt, until the first nan or inf value shows up in the graph. This is analogous to conditional breakpoints in some procedural-language debuggers:

tfdbg> run -f has\_inf\_or\_nan

NOTE: This works because we have previously registered a filter for nans and infs called has\_inf\_or\_nan(as explained previously). If you have registered any other filters, you can let **tfdbg** run till any tensors pass that filter as well, e.g.,

# In python code:  
sess.add\_tensor\_filter('my\_filter', my\_filter\_callable)  
  
# Run at tfdbg run-start prompt:  
tfdbg> run -f my\_filter

After you enter run -f has\_inf\_or\_nan, you will see the following screen with a red-colored title line indicating **tfdbg** stopped immediately after a run() call generated intermediate tensors that passed the specified filter has\_inf\_or\_nan:



As the screen display indicates, the has\_inf\_or\_nan filter is first passed during the fourth run() call: an [Adam optimizer](https://arxiv.org/abs/1412.6980) forward-backward training pass on the graph. In this run, 36 (out of the total 95) intermediate tensors contain nan or inf values. These tensors are listed in chronological order, with their timestamps displayed on the left. At the top of the list, you can see the first tensor in which the bad numerical values first surfaced: cross\_entropy/Log:0.

To view the value of the tensor, click the underlined tensor name cross\_entropy/Log:0 or enter the equivalent command:

tfdbg> pt cross\_entropy/Log:0

Scroll down a little and you will notice some scattered inf values. If the instances of inf and nan are difficult to spot by eye, you can use the following command to perform a regex search and highlight the output:

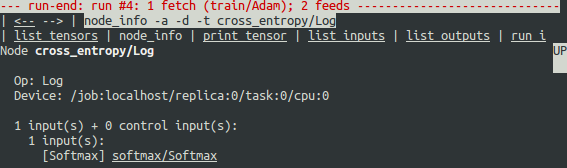
tfdbg> /inf

Or, alternatively:

tfdbg> /(inf|nan)

Why did these infinities appear? To further debug, display more information about the node cross\_entropy/Logby clicking the underlined node\_info menu item on the top or entering the equivalent command:

tfdbg> ni cross\_entropy/Log



You can see that this node has the op type Log and that its input is the node softmax/Softmax. Run the following command to take a closer look at the input tensor:

tfdbg> pt softmax/Softmax:0

Examine the values in the input tensor, and search to see if there are any zeros:

tfdbg> /0\.000

Indeed, there are zeros. Now it is clear that the origin of the bad numerical values is the node cross\_entropy/Log taking logs of zeros. To find out the culprit line in the Python source code, use the -t flag of the ni command to show the traceback of the node's construction:

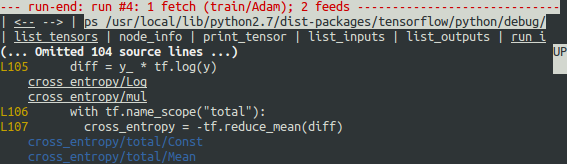
tfdbg> ni -t cross\_entropy/Log

The -t flag is used by default, if you use the clickable "node\_info" menu item at the top of the screen.

From the traceback, you can see that the op is constructed around line 106 of [debug\_mnist.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/python/debug/examples/debug_mnist.py):

diff = y\_ \* tf.log(y)

**\*tfdbg** has a feature that makes it ease to trace Tensors and ops back to lines in Python source files. It can annotate lines of a Python file with the ops or Tensors created by them. To use this feature, simply click the underlined line numbers in the stack trace output of the ni -t <op\_name> commands, or use the ps (or print\_source) command such as: ps /path/to/source.py. See the screenshot below for an example of psoutput:



Apply a value clipping on the input to [tf.log](https://www.tensorflow.org/api_docs/python/tf/log) to resolve this problem:

diff = y\_ \* tf.log(tf.clip\_by\_value(y, 1e-8, 1.0))

Now, try training again with --debug:

python -m tensorflow.python.debug.examples.debug\_mnist --debug

Enter run -f has\_inf\_or\_nan at the tfdbg> prompt and confirm that no tensors are flagged as containing nan or inf values, and accuracy no longer gets stuck. Success!

## Debugging tf-learn Estimators

For documentation on **tfdbg** to debug [tf.contrib.learn](https://www.tensorflow.org/get_started/tflearn) Estimators and Experiments, please see [How to Use TensorFlow Debugger (tfdbg) with tf.contrib.learn](https://www.tensorflow.org/programmers_guide/tfdbg-tflearn).

## Offline Debugging of Remotely-Running Sessions

Oftentimes, your model is running in a remote machine or process that you don't have terminal access to. To perform model debugging in such cases, you can use the offline\_analyzer of tfdbg. It operates on dumped data directories. If the process you are running is written in Python, you can configure the RunOptions proto that you call your Session.run() method with, by using the method [tfdbg.watch\_graph](https://www.tensorflow.org/api_docs/python/tfdbg/watch_graph). This will cause the intermediate tensors and runtime graphs to be dumped to a shared storage location of your choice when the Session.run() call occurs. For example:

from tensorflow.python.debug import debug\_utils  
  
# ... Code where your session and graph are set up...  
  
run\_options = tf.RunOptions()  
debug\_utils.watch\_graph(  
      run\_options,  
      session.graph,  
      debug\_urls=["file:///shared/storage/location/tfdbg\_dumps\_1"])  
# Be sure to use different directories for different run() calls.  
  
session.run(fetches, feed\_dict=feeds, options=run\_options)

Later, in an environment that you have terminal access to, you can load and inspect the data in the dump directory on the shared storage by using the offline\_analyzer of tfdbg. For example:

python -m tensorflow.python.debug.cli.offline\_analyzer \  
    --dump\_dir=/shared/storage/location/tfdbg\_dumps\_1

The Session wrapper DumpingDebugWrapperSession offers an easier and more flexible way to generate dumps on filesystem that can be analyzed offline. To use it, simply do:

# Let your BUILD target depend on "//tensorflow/python/debug:debug\_py  
# (You don't need to worry about the BUILD dependency if you are using a pip  
#  install of open-source TensorFlow.)  
from tensorflow.python.debug import debug\_utils  
  
sess = tf\_debug.DumpingDebugWrapperSession(  
    sess, "/shared/storage/location/tfdbg\_dumps\_1/", watch\_fn=my\_watch\_fn)

watch\_fn=my\_watch\_fn is a Callable that allows you to configure what Tensors to watch on different Session.run() calls, as a function of the fetches and feed\_dict to the run() call and other states. See [the API doc of DumpingDebugWrapperSession](https://www.tensorflow.org/api_docs/python/tfdbg/DumpingDebugWrapperSession#__init__) for more details.

If you model code is written in C++ or other languages, you can also modify the debug\_options field of RunOptions to generate debug dumps that can be inspected offline. See [the proto definition](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/debug.proto) for more details.

## Other Features of the tfdbg CLI

* Navigation through command history using the Up and Down arrow keys. Prefix-based navigation is also supported.
* Navigation through history of screen outputs using the prev and next commands or by clicking the underlined <-- and --> links near the top of the screen.
* Tab completion of commands and some command arguments.
* Write screen output to file by using bash-style redirection. For example:

tfdbg> pt cross\_entropy/Log:0[:, 0:10] > /tmp/xent\_value\_slices.txt

## Frequently Asked Questions

**Q**: Do the timestamps on the left side of theltoutput reflect actual performance in a non-debugging session?

**A**: No. The debugger inserts additional special-purpose debug nodes to the graph to record the values of intermediate tensors. These nodes certainly slow down the graph execution. If you are interested in profiling your model, check out [tfprof](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/tfprof) and other profiling tools for TensorFlow.

**Q**: How do I link tfdbg against mySessionin Bazel? Why do I see an error such as "ImportError: cannot import name debug"?

**A**: In your BUILD rule, declare dependencies: "//tensorflow:tensorflow\_py" and "//tensorflow/python/debug:debug\_py". The first is the dependency that you include to use TensorFlow even without debugger support; the second enables the debugger. Then, In your Python file, add:

from tensorflow.python import debug as tf\_debug  
  
# Then wrap your TensorFlow Session with the local-CLI wrapper.  
sess = tf\_debug.LocalCLIDebugWrapperSession(sess)

**Q**: Does tfdbg help debugging runtime errors such as shape mismatches?

**A**: Yes. tfdbg intercepts errors generated by ops during runtime and presents the errors with some debug instructions to the user in the CLI. See examples:

# Debugging shape mismatch during matrix multiplication.  
python -m tensorflow.python.debug.examples.debug\_errors \  
    --error shape\_mismatch --debug  
  
# Debugging uninitialized variable.  
python -m tensorflow.python.debug.examples.debug\_errors \  
    --error uninitialized\_variable --debug

**Q**: Why can't I select text in the tfdbg CLI?

**A**: This is because the tfdbg CLI enables mouse events in the terminal by default. This [mouse-mask](https://linux.die.net/man/3/mousemask) mode overrides default terminal interactions, including text selection. You can re-enable text selection by using the command mouse off or m off.

**Q**: What are the platform-specific system requirements of***tfdbg***CLI in open-source TensorFlow?

**A**: On Mac OS X, the ncurses library is required. It can be installed with brew install homebrew/dupes/ncurses. On Windows, the pyreadline library is required. If you are using Anaconda3, you can install it with a command such as "C:\Program Files\Anaconda3\Scripts\pip.exe" install pyreadline.

# How to Use TensorFlow Debugger (tfdbg) with tf.contrib.learn

In [a previous tutorial](https://www.tensorflow.org/programmers_guide/debugger), we described how to use TensorFlow Debugger (**tfdbg**) to debug TensorFlow graphs running in [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) objects managed by yourself. However, many users find [tf.contrib.learn](https://www.tensorflow.org/get_started/tflearn) [Estimator](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/Estimator)s to be a convenient higher-level API for creating and using models in TensorFlow. Part of the convenience is that Estimators manage Sessions internally. Fortunately, you can still use tfdbg with Estimators by adding special hooks.

## Debugging tf.contrib.learn Estimators

Currently, **tfdbg** can debug the [fit()](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/BaseEstimator#fit) [evaluate()](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/BaseEstimator#evaluate) methods of tf-learn Estimators. To debug Estimator.fit(), create a LocalCLIDebugHook and supply it as the monitors argument. For example:

# First, let your BUILD target depend on "//tensorflow/python/debug:debug\_py"  
# (You don't need to worry about the BUILD dependency if you are using a pip  
#  install of open-source TensorFlow.)  
from tensorflow.python import debug as tf\_debug  
  
hooks = [tf\_debug.LocalCLIDebugHook()]  
  
# Create a local CLI debug hook and use it as a monitor when calling fit().  
classifier.fit(x=training\_set.data,  
               y=training\_set.target,  
               steps=1000,  
               monitors=hooks)

To debug Estimator.evaluate(), you can follow the example below:

accuracy\_score = classifier.evaluate(x=test\_set.data,  
                                     y=test\_set.target,  
                                     hooks=hooks)["accuracy"]

For a detailed [example](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/python/debug/examples/debug_tflearn_iris.py) based on [tf-learn's iris tutorial](https://www.tensorflow.org/get_started/tflearn), run:

python -m tensorflow.python.debug.examples.debug\_tflearn\_iris --debug

## Debugging tf.contrib.learn Experiments

Experiment is a construct in tf.contrib.learn at a higher level than Estimator. It provides a single interface for training and evaluating a model. To debug the train() and evaluate() calls to an Experimentobject, you can use the keyword arguments train\_monitors and eval\_hooks, respectively, when calling its constructor. For example:

# First, let your BUILD target depend on "//tensorflow/python/debug:debug\_py"  
# (You don't need to worry about the BUILD dependency if you are using a pip  
#  install of open-source TensorFlow.)  
from tensorflow.python import debug as tf\_debug  
  
hooks = [tf\_debug.LocalCLIDebugHook()]  
  
ex = experiment.Experiment(classifier,  
                           train\_input\_fn=iris\_input\_fn,  
                           eval\_input\_fn=iris\_input\_fn,  
                           train\_steps=FLAGS.train\_steps,  
                           eval\_delay\_secs=0,  
                           eval\_steps=1,  
                           train\_monitors=hooks,  
                           eval\_hooks=hooks)  
  
ex.train()  
accuracy\_score = ex.evaluate()["accuracy"]

To see the debug\_tflearn\_iris example run in the Experiment mode, do:

python -m tensorflow.python.debug.examples.debug\_tflearn\_iris \  
    --use\_experiment --debug

## Debugging Estimators and Experiments without Terminal Access

If your Estimator or Experiment is running in an environment to which you do not have command-line access (e.g., a remote server), you can use the non-interactive DumpingDebugHook. For example:

# Let your BUILD target depend on "//tensorflow/python/debug:debug\_py  
# (You don't need to worry about the BUILD dependency if you are using a pip  
#  install of open-source TensorFlow.)  
from tensorflow.python import debug as tf\_debug  
  
hooks = [tf\_debug.DumpingDebugHook("/shared/storage/location/tfdbg\_dumps\_1")]

Then this hook can be used in the same way as the LocalCLIDebugHook examples above. As the training and/or evalution of Estimator or Experiment happens, directories of the naming pattern/shared/storage/location/tfdbg\_dumps\_1/run\_<epoch\_timestamp\_microsec>\_<uuid> will appear. Each directory corresponds to a Session.run() call that underlies the fit() or evaluate() call. You can load these directories and inspect them in a command-line interface in an offline manner using theoffline\_analyzer offered by **tfdbg**. For example:

python -m tensorflow.python.debug.cli.offline\_analyzer \  
    --dump\_dir="/shared/storage/location/tfdbg\_dumps\_1/run\_<epoch\_timestamp\_microsec>\_<uuid>"

The LocalCLIDebugHook also allows you to configure a watch\_fn that can be used to flexibly specify what Tensors to watch on different Session.run() calls, as a function of the fetches and feed\_dict and other states. See [this API doc](https://www.tensorflow.org/api_docs/python/tfdbg/DumpingDebugWrapperSession#__init__) for more details.

# Exporting and Importing a MetaGraph

A [MetaGraph](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/meta_graph.proto) contains both a TensorFlow GraphDef as well as associated metadata necessary for running computation in a graph when crossing a process boundary. It can also be used for long term storage of graphs. The MetaGraph contains the information required to continue training, perform evaluation, or run inference on a previously trained graph.

The APIs for exporting and importing the complete model are in the [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver) class:[tf.train.export\_meta\_graph](https://www.tensorflow.org/api_docs/python/tf/train/export_meta_graph) and [tf.train.import\_meta\_graph](https://www.tensorflow.org/api_docs/python/tf/train/import_meta_graph).

## What's in a MetaGraph

The information contained in a MetaGraph is expressed as a [MetaGraphDef](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/meta_graph.proto) protocol buffer. It contains the following fields:

* [MetaInfoDef](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/meta_graph.proto) for meta information, such as version and other user information.
* [GraphDef](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/framework/graph.proto) for describing the graph.
* [SaverDef](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/saver.proto) for the saver.
* [CollectionDef](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/core/protobuf/meta_graph.proto) map that further describes additional components of the model, such as[Variables](https://www.tensorflow.org/api_guides/python/state_ops), [tf.train.QueueRunner](https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner), etc. In order for a Python object to be serialized to and from MetaGraphDef, the Python class must implement to\_proto() andfrom\_proto() methods, and register them with the system usingregister\_proto\_function.

For example,

def to\_proto(self, export\_scope=None):  
  
  """Converts a `Variable` to a `VariableDef` protocol buffer.  
  
  Args:  
    export\_scope: Optional `string`. Name scope to remove.  
  
  Returns:  
    A `VariableDef` protocol buffer, or `None` if the `Variable` is not  
    in the specified name scope.  
  """  
  if (export\_scope is None or  
      self.\_variable.name.startswith(export\_scope)):  
    var\_def = variable\_pb2.VariableDef()  
    var\_def.variable\_name = ops.strip\_name\_scope(  
        self.\_variable.name, export\_scope)  
    var\_def.initializer\_name = ops.strip\_name\_scope(  
        self.initializer.name, export\_scope)  
    var\_def.snapshot\_name = ops.strip\_name\_scope(  
        self.\_snapshot.name, export\_scope)  
    if self.\_save\_slice\_info:  
      var\_def.save\_slice\_info\_def.MergeFrom(self.\_save\_slice\_info.to\_proto(  
          export\_scope=export\_scope))  
    return var\_def  
  else:  
    return None  
  
@staticmethod  
def from\_proto(variable\_def, import\_scope=None):  
  """Returns a `Variable` object created from `variable\_def`."""  
  return Variable(variable\_def=variable\_def, import\_scope=import\_scope)  
  
ops.register\_proto\_function(ops.GraphKeys.GLOBAL\_VARIABLES,  
                            proto\_type=variable\_pb2.VariableDef,  
                            to\_proto=Variable.to\_proto,  
                            from\_proto=Variable.from\_proto)

## Exporting a Complete Model to MetaGraph

The API for exporting a running model as a MetaGraph is export\_meta\_graph().

def export\_meta\_graph(filename=None, collection\_list=None, as\_text=False):  
  """Writes `MetaGraphDef` to save\_path/filename.  
  
  Args:  
    filename: Optional meta\_graph filename including the path.  
    collection\_list: List of string keys to collect.  
    as\_text: If `True`, writes the meta\_graph as an ASCII proto.  
  
  Returns:  
    A `MetaGraphDef` proto.  
  """

A collection can contain any Python objects that users would like to be able to uniquely identify and easily retrieve. These objects can be special operations in the graph, such as train\_op, or hyper parameters, such as "learning rate". Users can specify the list of collections they would like to export. If no collection\_list is specified, all collections in the model will be exported.

The API returns a serialized protocol buffer. If filename is specified, the protocol buffer will also be written to a file.

Here are some of the typical usage models:

* Export the default running graph:

# Build the model  
...  
with tf.Session() as sess:  
  # Use the model  
  ...  
# Export the model to /tmp/my-model.meta.  
meta\_graph\_def = tf.train.export\_meta\_graph(filename='/tmp/my-model.meta')

* Export the default running graph and only a subset of the collections.

meta\_graph\_def = tf.train.export\_meta\_graph(  
    filename='/tmp/my-model.meta',  
    collection\_list=["input\_tensor", "output\_tensor"])

The MetaGraph is also automatically exported via the save() API in [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver).

## Import a MetaGraph

The API for importing a MetaGraph file into a graph is import\_meta\_graph().

Here are some of the typical usage models:

* Import and continue training without building the model from scratch.

...  
# Create a saver.  
saver = tf.train.Saver(...variables...)  
# Remember the training\_op we want to run by adding it to a collection.  
tf.add\_to\_collection('train\_op', train\_op)  
sess = tf.Session()  
for step in xrange(1000000):  
    sess.run(train\_op)  
    if step % 1000 == 0:  
        # Saves checkpoint, which by default also exports a meta\_graph  
        # named 'my-model-global\_step.meta'.  
        saver.save(sess, 'my-model', global\_step=step)

Later we can continue training from this saved meta\_graph without building the model from scratch.

with tf.Session() as sess:  
  new\_saver = tf.train.import\_meta\_graph('my-save-dir/my-model-10000.meta')  
  new\_saver.restore(sess, 'my-save-dir/my-model-10000')  
  # tf.get\_collection() returns a list. In this example we only want the  
  # first one.  
  train\_op = tf.get\_collection('train\_op')[0]  
  for step in xrange(1000000):  
    sess.run(train\_op)

* Import and extend the graph.

For example, we can first build an inference graph, export it as a meta graph:

# Creates an inference graph.  
# Hidden 1  
images = tf.constant(1.2, tf.float32, shape=[100, 28])  
with tf.name\_scope("hidden1"):  
  weights = tf.Variable(  
      tf.truncated\_normal([28, 128],  
                          stddev=1.0 / math.sqrt(float(28))),  
      name="weights")  
  biases = tf.Variable(tf.zeros([128]),  
                       name="biases")  
  hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)  
# Hidden 2  
with tf.name\_scope("hidden2"):  
  weights = tf.Variable(  
      tf.truncated\_normal([128, 32],  
                          stddev=1.0 / math.sqrt(float(128))),  
      name="weights")  
  biases = tf.Variable(tf.zeros([32]),  
                       name="biases")  
  hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)  
# Linear  
with tf.name\_scope("softmax\_linear"):  
  weights = tf.Variable(  
      tf.truncated\_normal([32, 10],  
                          stddev=1.0 / math.sqrt(float(32))),  
      name="weights")  
  biases = tf.Variable(tf.zeros([10]),  
                       name="biases")  
  logits = tf.matmul(hidden2, weights) + biases  
  tf.add\_to\_collection("logits", logits)  
  
init\_all\_op = tf.global\_variables\_initializer()  
  
with tf.Session() as sess:  
  # Initializes all the variables.  
  sess.run(init\_all\_op)  
  # Runs to logit.  
  sess.run(logits)  
  # Creates a saver.  
  saver0 = tf.train.Saver()  
  saver0.save(sess, 'my-save-dir/my-model-10000')  
  # Generates MetaGraphDef.  
  saver0.export\_meta\_graph('my-save-dir/my-model-10000.meta')

Then later import it and extend it to a training graph.

with tf.Session() as sess:  
  new\_saver = tf.train.import\_meta\_graph('my-save-dir/my-model-10000.meta')  
  new\_saver.restore(sess, 'my-save-dir/my-model-10000')  
  # Addes loss and train.  
  labels = tf.constant(0, tf.int32, shape=[100], name="labels")  
  batch\_size = tf.size(labels)  
  labels = tf.expand\_dims(labels, 1)  
  indices = tf.expand\_dims(tf.range(0, batch\_size), 1)  
  concated = tf.concat([indices, labels], 1)  
  onehot\_labels = tf.sparse\_to\_dense(  
      concated, tf.stack([batch\_size, 10]), 1.0, 0.0)  
  logits = tf.get\_collection("logits")[0]  
  cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(  
      labels=onehot\_labels, logits=logits, name="xentropy")  
  loss = tf.reduce\_mean(cross\_entropy, name="xentropy\_mean")  
  
  tf.summary.scalar('loss', loss)  
  # Creates the gradient descent optimizer with the given learning rate.  
  optimizer = tf.train.GradientDescentOptimizer(0.01)  
  
  # Runs train\_op.  
  train\_op = optimizer.minimize(loss)  
  sess.run(train\_op)

* Import a graph with preset devices.

Sometimes an exported meta graph is from a training environment that the importer doesn't have. For example, the model might have been trained on GPUs, or in a distributed environment with replicas. When importing such models, it's useful to be able to clear the device settings in the graph so that we can run it on locally available devices. This can be achieved by calling import\_meta\_graph with the clear\_devices option set to True.

with tf.Session() as sess:  
  new\_saver = tf.train.import\_meta\_graph('my-save-dir/my-model-10000.meta',  
      clear\_devices=True)  
  new\_saver.restore(sess, 'my-save-dir/my-model-10000')  
  ...

* Import within the default graph.

Sometimes you might want to run export\_meta\_graph and import\_meta\_graph in codelab using the default graph. In that case, you need to reset the default graph by calling tf.reset\_default\_graph() first before running import.

meta\_graph\_def = tf.train.export\_meta\_graph()  
...  
tf.reset\_default\_graph()  
...  
tf.train.import\_meta\_graph(meta\_graph\_def)  
...

* Retrieve Hyper Parameters

filename = ".".join([tf.train.latest\_checkpoint(train\_dir), "meta"])  
tf.train.import\_meta\_graph(filename)  
hparams = tf.get\_collection("hparams")

# TensorFlow Version Semantics

## Semantic Versioning 2.0

TensorFlow follows Semantic Versioning 2.0 ([semver](http://semver.org/)) for its public API. Each release version of TensorFlow has the form MAJOR.MINOR.PATCH. Changes to the each number have the following meaning:

* **MAJOR**: Backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with a new release. However, in some cases existing TensorFlow data (graphs, checkpoints, and other protobufs) may be migratable to the newer release; see below for details on data compatibility.
* **MINOR**: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release and which depends only the public API will continue to work unchanged. For details on what is and is not the public API, see below.
* **PATCH**: Backwards compatible bug fixes.

## What is covered

Only the public APIs of TensorFlow are backwards compatible across minor and patch versions. The public APIs consist of

* The documented public [Python](https://www.tensorflow.org/api_docs/python) API, excluding tf.contrib. This includes all public functions and classes (whose names do not start with \_) in the tensorflow module and its submodules. Note that the code in the examples/ to tools/ directories is not reachable through the tensorflow Python module and is thus not covered by the compatibility guarantee.

If a symbol is available through the tensorflow Python module or its submodules, but is not documented, then it is not considered part of the public API.

* The [C API](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h).
* The following protocol buffer files: [attr\_value](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/attr_value.proto), [config](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto), [event](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/event.proto), [graph](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/graph.proto), [op\_def](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_def.proto), [reader\_base](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/reader_base.proto),[summary](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/summary.proto), [tensor](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor.proto), [tensor\_shape](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor_shape.proto), and [types](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.proto).

## What is not covered

Some API functions are explicitly marked as "experimental" and can change in backward incompatible ways between minor releases. These include:

* **Experimental APIs**: The [tf.contrib](https://www.tensorflow.org/api_docs/python/tf/contrib) module and its submodules in Python and any functions in the C API or fields in protocol buffers that are explicitly commented as being experimental.
* **Other languages**: TensorFlow APIs in languages other than Python and C, such as:
* [C++](https://www.tensorflow.org/api_guides/cc/guide) (exposed through header files in [tensorflow/cc](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/cc)).
* [Java](https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/package-summary), and
* [Go](https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go)
* **Details of composite ops:** Many public functions in Python expand to several primitive ops in the graph, and these details will be part of any graphs saved to disk as GraphDefs. These details are allowed to change for minor releases. In particular, regressions tests that check for exact matching between graphs are likely to break across minor releases, even though the behavior of the graph should be unchanged and existing checkpoints will still work.
* **Floating point numerical details:** The specific floating point values computed by ops may change at any time: users should rely only on approximate accuracy and numerical stability, not on the specific bits computed. Changes to numerical formulas in minor and patch releases should result in comparable or improved accuracy, with the caveat that in machine learning improved accuracy of specific formulas may result in worse accuracy for the overall system.
* **Random numbers:** The specific random numbers computed by the [random ops](https://www.tensorflow.org/api_guides/python/constant_op#Random_Tensors) may change at any time: users should rely only on approximately correct distributions and statistical strength, not the specific bits computed. However, we will make changes to random bits rarely and ideally never for patch releases, and all such intended changes will be documented.
* **Distributed Tensorflow:** Running 2 different versions of TensorFlow in a single cluster is unsupported. There are no guarantees about backwards compatibility of the wire protocol.

Furthermore, any API methods marked "deprecated" in the 1.0 release can be deleted in any subsequent minor release.

## Compatibility for Graphs and Checkpoints

Many users of TensorFlow will be saving graphs and trained models to disk for later evaluation or more training, often changing versions of TensorFlow in the process. First, following semver, any graph or checkpoint written out with one version of TensorFlow can be loaded and evaluated with a later version of TensorFlow with the same major release. However, we will endeavour to preserve backwards compatibility even across major releases when possible, so that the serialized files are usable over long periods of time.

There are two main classes of saved TensorFlow data: graphs and checkpoints. Graphs describe the data flow graphs of ops to be run during training and inference, and checkpoints contain the saved tensor values of variables in a graph.

Graphs are serialized via the GraphDef protocol buffer. To facilitate (rare) backwards incompatible changes to graphs, each GraphDef has an integer version separate from the TensorFlow version. The semantics are:

* Each version of TensorFlow supports an interval of GraphDef versions. This interval with be constant across patch releases, and will only grow across minor releases. Dropping support for a GraphDef version will only occur for a major release of TensorFlow.
* Newly created graphs use the newest GraphDef version.
* If a given version of TensorFlow supports the GraphDef version of a graph, it will load and evaluate with the same behavior as when it was written out (except for floating point numerical details and random numbers), regardless of the major version of TensorFlow. In particular, all checkpoint files will be compatible.
* If the GraphDef upper bound is increased to X in a (minor) release, there will be at least six months before the lower bound is increased to X.

For example (numbers and versions hypothetical), TensorFlow 1.2 might support GraphDef versions 4 to 7. TensorFlow 1.3 could add GraphDef version 8 and support versions 4 to 8. At least six months later, TensorFlow 2.0.0 could drop support for versions 4 to 7, leaving version 8 only.

Finally, when support for a GraphDef version is dropped, we will attempt to provide tools for automatically converting graphs to a newer supported GraphDef version.

For developer-level details about GraphDef versioning, including how to evolve the versions to account for changes, see [TensorFlow Data Versioning](https://www.tensorflow.org/programmers_guide/data_versions).

# TensorFlow Version Semantics

## Semantic Versioning 2.0

TensorFlow follows Semantic Versioning 2.0 ([semver](http://semver.org/)) for its public API. Each release version of TensorFlow has the form MAJOR.MINOR.PATCH. Changes to the each number have the following meaning:

* **MAJOR**: Backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with a new release. However, in some cases existing TensorFlow data (graphs, checkpoints, and other protobufs) may be migratable to the newer release; see below for details on data compatibility.
* **MINOR**: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release and which depends only the public API will continue to work unchanged. For details on what is and is not the public API, see below.
* **PATCH**: Backwards compatible bug fixes.

## What is covered

Only the public APIs of TensorFlow are backwards compatible across minor and patch versions. The public APIs consist of

* The documented public [Python](https://www.tensorflow.org/api_docs/python) API, excluding tf.contrib. This includes all public functions and classes (whose names do not start with \_) in the tensorflow module and its submodules. Note that the code in the examples/ to tools/ directories is not reachable through the tensorflow Python module and is thus not covered by the compatibility guarantee.

If a symbol is available through the tensorflow Python module or its submodules, but is not documented, then it is not considered part of the public API.

* The [C API](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h).
* The following protocol buffer files: [attr\_value](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/attr_value.proto), [config](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto), [event](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/event.proto), [graph](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/graph.proto), [op\_def](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_def.proto), [reader\_base](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/reader_base.proto),[summary](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/summary.proto), [tensor](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor.proto), [tensor\_shape](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor_shape.proto), and [types](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.proto).

## What is not covered

Some API functions are explicitly marked as "experimental" and can change in backward incompatible ways between minor releases. These include:

* **Experimental APIs**: The [tf.contrib](https://www.tensorflow.org/api_docs/python/tf/contrib) module and its submodules in Python and any functions in the C API or fields in protocol buffers that are explicitly commented as being experimental.
* **Other languages**: TensorFlow APIs in languages other than Python and C, such as:
* [C++](https://www.tensorflow.org/api_guides/cc/guide) (exposed through header files in [tensorflow/cc](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/cc)).
* [Java](https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/package-summary), and
* [Go](https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go)
* **Details of composite ops:** Many public functions in Python expand to several primitive ops in the graph, and these details will be part of any graphs saved to disk as GraphDefs. These details are allowed to change for minor releases. In particular, regressions tests that check for exact matching between graphs are likely to break across minor releases, even though the behavior of the graph should be unchanged and existing checkpoints will still work.
* **Floating point numerical details:** The specific floating point values computed by ops may change at any time: users should rely only on approximate accuracy and numerical stability, not on the specific bits computed. Changes to numerical formulas in minor and patch releases should result in comparable or improved accuracy, with the caveat that in machine learning improved accuracy of specific formulas may result in worse accuracy for the overall system.
* **Random numbers:** The specific random numbers computed by the [random ops](https://www.tensorflow.org/api_guides/python/constant_op#Random_Tensors) may change at any time: users should rely only on approximately correct distributions and statistical strength, not the specific bits computed. However, we will make changes to random bits rarely and ideally never for patch releases, and all such intended changes will be documented.
* **Distributed Tensorflow:** Running 2 different versions of TensorFlow in a single cluster is unsupported. There are no guarantees about backwards compatibility of the wire protocol.

Furthermore, any API methods marked "deprecated" in the 1.0 release can be deleted in any subsequent minor release.

## Compatibility for Graphs and Checkpoints

Many users of TensorFlow will be saving graphs and trained models to disk for later evaluation or more training, often changing versions of TensorFlow in the process. First, following semver, any graph or checkpoint written out with one version of TensorFlow can be loaded and evaluated with a later version of TensorFlow with the same major release. However, we will endeavour to preserve backwards compatibility even across major releases when possible, so that the serialized files are usable over long periods of time.

There are two main classes of saved TensorFlow data: graphs and checkpoints. Graphs describe the data flow graphs of ops to be run during training and inference, and checkpoints contain the saved tensor values of variables in a graph.

Graphs are serialized via the GraphDef protocol buffer. To facilitate (rare) backwards incompatible changes to graphs, each GraphDef has an integer version separate from the TensorFlow version. The semantics are:

* Each version of TensorFlow supports an interval of GraphDef versions. This interval with be constant across patch releases, and will only grow across minor releases. Dropping support for a GraphDef version will only occur for a major release of TensorFlow.
* Newly created graphs use the newest GraphDef version.
* If a given version of TensorFlow supports the GraphDef version of a graph, it will load and evaluate with the same behavior as when it was written out (except for floating point numerical details and random numbers), regardless of the major version of TensorFlow. In particular, all checkpoint files will be compatible.
* If the GraphDef upper bound is increased to X in a (minor) release, there will be at least six months before the lower bound is increased to X.

For example (numbers and versions hypothetical), TensorFlow 1.2 might support GraphDef versions 4 to 7. TensorFlow 1.3 could add GraphDef version 8 and support versions 4 to 8. At least six months later, TensorFlow 2.0.0 could drop support for versions 4 to 7, leaving version 8 only.

Finally, when support for a GraphDef version is dropped, we will attempt to provide tools for automatically converting graphs to a newer supported GraphDef version.

For developer-level details about GraphDef versioning, including how to evolve the versions to account for changes, see [TensorFlow Data Versioning](https://www.tensorflow.org/programmers_guide/data_versions).

# TensorFlow Data Versioning: GraphDefs and Checkpoints

As described in [Compatibility for Graphs and Checkpoints](https://www.tensorflow.org/programmers_guide/version_semantics#compatibility_for_graphs_and_checkpoints), TensorFlow marks each kind of data with version information in order to maintain backward compatibility. This document provides additional details about the versioning mechanism, and how to use it to safely change data formats.

## Backward and partial forward compatibility

The two core artifacts exported from and imported into TensorFlow are checkpoints (serialized variable states) and GraphDefs (serialized computation graphs). Any approach to versioning these artifacts must take into account the following requirements:

* **Backward compatibility** to support loading GraphDefs created with older versions of TensorFlow.
* **Forward compatibility** to support scenarios where the producer of a GraphDef is upgraded to a newer version of TensorFlow before the consumer.
* Enable evolving TensorFlow in incompatible ways. For example, removing Ops, adding attributes, and removing attributes.

For GraphDefs, backward compatibility is enforced within a major version. This means functionality can only be removed between major versions. Forward compatibility is enforced within Patch releases (1.x.1 -> 1.x.2, for example).

In order to achieve backward and forward compatibility as well as know when to enforce changes in formats, the serialized representations of graphs and variable state need to have metadata that describes when they were produced. The sections below detail the TensorFlow implementation and guidelines for evolving GraphDefversions.

### Independent data version schemes

There are data versions for GraphDefs and checkpoints. Both data formats evolve at different rates, and also at different speeds than the version of TensorFlow. Both versioning systems are defined in[core/public/version.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/public/version.h). Whenever a new version is added a note is added to the header detailing what changed and the date.

### Data, producers, and consumers

This section discusses version information for **data**, binaries that produce data (**producers**), and binaries that consume data (**consumers**):

* Producer binaries have a version (producer) and a minimum consumer version that they are compatible with (min\_consumer).
* Consumer binaries have a version (consumer) and a minimum producer version that they are compatible with (min\_producer).
* Each piece of versioned data has a [VersionDef versions](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/versions.proto) field which records the producer that made the data, the min\_consumer that it is compatible with, and a list of bad\_consumers versions that are disallowed.

By default, when a producer makes some data, the data inherits the producer's producer and min\_consumerversions. bad\_consumers can be set if specific consumer versions are known to contain bugs and must be avoided. A consumer can accept a piece of data if

* consumer >= data's min\_consumer
* data's producer >= consumer's min\_producer
* consumer not in data's bad\_consumers

Since both producers and consumers come from the same TensorFlow code base, [core/public/version.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/public/version.h)contains a main binary version which is treated as either producer or consumer depending on context and both min\_consumer and min\_producer (needed by producers and consumers, respectively). Specifically,

* For GraphDef versions, we have TF\_GRAPH\_DEF\_VERSION, TF\_GRAPH\_DEF\_VERSION\_MIN\_CONSUMER, andTF\_GRAPH\_DEF\_VERSION\_MIN\_PRODUCER.
* For checkpoint versions, we have TF\_CHECKPOINT\_VERSION, TF\_CHECKPOINT\_VERSION\_MIN\_CONSUMER, and TF\_CHECKPOINT\_VERSION\_MIN\_PRODUCER.

### Evolving GraphDef versions

This section presents examples of using this versioning mechanism to make changes to the GraphDef format.

**Adding a new Op:**

1. Add the new Op to both consumers and producers at the same time, and do not change any GraphDefversions. This type of change is automatically backward compatible, and does not impact forward compatibility plan since existing producer scripts will not suddenly use the new functionality.

**Adding a new Op and switching existing Python wrappers to use it:**

1. Implement new consumer functionality and increment the binary version.
2. If it is possible to make the wrappers use the new functionality only in cases that did not work before, the wrappers can be updated now.
3. Change Python wrappers to use the new functionality. Do not increment min\_consumer, since models which do not use this Op should not break.

**Removing an Op or restricting the functionality of an Op:**

1. Fix all producer scripts (not TensorFlow itself) to not use the banned Op or functionality.
2. Increment the binary version and implement new consumer functionality that bans the removed Op or functionality for GraphDefs at the new version and above. If possible, make TensorFlow stop producing GraphDefs with the banned functionality. This can be done with[REGISTER\_OP(...).Deprecated(deprecated\_at\_version, message)](https://github.com/tensorflow/tensorflow/blob/b289bc7a50fc0254970c60aaeba01c33de61a728/tensorflow/core/ops/array_ops.cc#L1009).
3. Wait for a major release for backward compatibility purposes.
4. Increase min\_producer to the GraphDef version from (2) and remove the functionality entirely.

**Changing the functionality of an Op:**

1. Add a new similar Op named SomethingV2 or similar and go through the process of adding it and switching existing Python wrappers to use it (may take 3 weeks if forward compatibility is desired).
2. Remove the old Op (Can only take place with a major version change due to backward compatibility).
3. Increase min\_consumer to rule out consumers with the old Op, add back the old Op as an alias for SomethingV2, and go through the process to switch existing Python wrappers to use it.
4. Go through the process to remove SomethingV2.

**Banning a single consumer version that cannot run safely:**

1. Bump the binary version and add the bad version to bad\_consumers for all new GraphDefs. If possible, add to bad\_consumers only for GraphDefs which contain a certain Op or similar.
2. If existing consumers have the bad version, push them out as soon as possible.

# Frequently Asked Questions

This document provides answers to some of the frequently asked questions about TensorFlow. If you have a question that is not covered here, you might find an answer on one of the TensorFlow [community resources](https://www.tensorflow.org/about/index).

## Features and Compatibility

#### Can I run distributed training on multiple computers?

Yes! TensorFlow gained [support for distributed computation](https://www.tensorflow.org/deploy/distributed) in version 0.8. TensorFlow now supports multiple devices (CPUs and GPUs) in one or more computers.

#### Does TensorFlow work with Python 3?

As of the 0.6.0 release timeframe (Early December 2015), we do support Python 3.3+.

## Building a TensorFlow graph

See also the [API documentation on building graphs](https://www.tensorflow.org/api_guides/python/framework).

#### Why does c = tf.matmul(a, b) not execute the matrix multiplication immediately?

In the TensorFlow Python API, a, b, and c are [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) objects. A Tensor object is a symbolic handle to the result of an operation, but does not actually hold the values of the operation's output. Instead, TensorFlow encourages users to build up complicated expressions (such as entire neural networks and its gradients) as a dataflow graph. You then offload the computation of the entire dataflow graph (or a subgraph of it) to a TensorFlow [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session), which is able to execute the whole computation much more efficiently than executing the operations one-by-one.

#### How are devices named?

The supported device names are "/device:CPU:0" (or "/cpu:0") for the CPU device, and "/device:GPU:i"(or "/gpu:i") for the ith GPU device.

#### How do I place operations on a particular device?

To place a group of operations on a device, create them within a [with tf.device(name):](https://www.tensorflow.org/api_docs/python/tf/device) context. See the how-to documentation on [using GPUs with TensorFlow](https://www.tensorflow.org/tutorials/using_gpu) for details of how TensorFlow assigns operations to devices, and the [CIFAR-10 tutorial](https://www.tensorflow.org/tutorials/deep_cnn) for an example model that uses multiple GPUs.

#### What are the different types of tensors that are available?

TensorFlow supports a variety of different data types and tensor shapes. See the [ranks, shapes, and types reference](https://www.tensorflow.org/programmers_guide/dims_types) for more details.

## Running a TensorFlow computation

See also the [API documentation on running graphs](https://www.tensorflow.org/api_guides/python/client).

#### What's the deal with feeding and placeholders?

Feeding is a mechanism in the TensorFlow Session API that allows you to substitute different values for one or more tensors at run time. The feed\_dict argument to [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) is a dictionary that maps [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor)objects to numpy arrays (and some other types), which will be used as the values of those tensors in the execution of a step.

Often, you have certain tensors, such as inputs, that will always be fed. The [tf.placeholder](https://www.tensorflow.org/api_docs/python/tf/placeholder) op allows you to define tensors that must be fed, and optionally allows you to constrain their shape as well. See the [beginners' MNIST tutorial](https://www.tensorflow.org/get_started/mnist/beginners) for an example of how placeholders and feeding can be used to provide the training data for a neural network.

#### What is the difference between Session.run() and Tensor.eval()?

If t is a [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) object, [tf.Tensor.eval](https://www.tensorflow.org/api_docs/python/tf/Tensor#eval) is shorthand for [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) (where sess is the current [tf.get\_default\_session](https://www.tensorflow.org/api_docs/python/tf/get_default_session). The two following snippets of code are equivalent:

# Using `Session.run()`.  
sess = tf.Session()  
c = tf.constant(5.0)  
print(sess.run(c))  
  
# Using `Tensor.eval()`.  
c = tf.constant(5.0)  
with tf.Session():  
  print(c.eval())

In the second example, the session acts as a [context manager](https://docs.python.org/2.7/reference/compound_stmts.html#with), which has the effect of installing it as the default session for the lifetime of the with block. The context manager approach can lead to more concise code for simple use cases (like unit tests); if your code deals with multiple graphs and sessions, it may be more straightforward to make explicit calls to Session.run().

#### Do Sessions have a lifetime? What about intermediate tensors?

Sessions can own resources, such as [tf.Variable](https://www.tensorflow.org/api_docs/python/tf/Variable), [tf.QueueBase](https://www.tensorflow.org/api_docs/python/tf/QueueBase), and [tf.ReaderBase](https://www.tensorflow.org/api_docs/python/tf/ReaderBase); and these resources can use a significant amount of memory. These resources (and the associated memory) are released when the session is closed, by calling [tf.Session.close](https://www.tensorflow.org/api_docs/python/tf/Session#close).

The intermediate tensors that are created as part of a call to [Session.run()](https://www.tensorflow.org/api_guides/python/client) will be freed at or before the end of the call.

#### Does the runtime parallelize parts of graph execution?

The TensorFlow runtime parallelizes graph execution across many different dimensions:

* The individual ops have parallel implementations, using multiple cores in a CPU, or multiple threads in a GPU.
* Independent nodes in a TensorFlow graph can run in parallel on multiple devices, which makes it possible to speed up [CIFAR-10 training using multiple GPUs](https://www.tensorflow.org/tutorials/deep_cnn).
* The Session API allows multiple concurrent steps (i.e. calls to [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) in parallel. This enables the runtime to get higher throughput, if a single step does not use all of the resources in your computer.

#### Which client languages are supported in TensorFlow?

TensorFlow is designed to support multiple client languages. Currently, the best-supported client language is [Python](https://www.tensorflow.org/api_docs/python/index). Experimental interfaces for executing and constructing graphs are also available for [C++](https://www.tensorflow.org/api_docs/cc/index.md), [Java](https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/package-summary.html) and [Go](https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go).

TensorFlow also has a [C-based client API](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/c/c_api.h) to help build support for more client languages. We invite contributions of new language bindings.

#### Does TensorFlow make use of all the devices (GPUs and CPUs) available on my machine?

TensorFlow supports multiple GPUs and CPUs. See the how-to documentation on [using GPUs with TensorFlow](https://www.tensorflow.org/tutorials/using_gpu)for details of how TensorFlow assigns operations to devices, and the [CIFAR-10 tutorial](https://www.tensorflow.org/tutorials/deep_cnn) for an example model that uses multiple GPUs.

Note that TensorFlow only uses GPU devices with a compute capability greater than 3.5.

#### Why does Session.run() hang when using a reader or a queue?

The [tf.ReaderBase](https://www.tensorflow.org/api_docs/python/tf/ReaderBase) and [tf.QueueBase](https://www.tensorflow.org/api_docs/python/tf/QueueBase) classes provide special operations that can block until input (or free space in a bounded queue) becomes available. These operations allow you to build sophisticated [input pipelines](https://www.tensorflow.org/programmers_guide/reading_data), at the cost of making the TensorFlow computation somewhat more complicated. See the how-to documentation for [using QueueRunner objects to drive queues and readers](https://www.tensorflow.org/programmers_guide/reading_data#creating_threads_to_prefetch_using_queuerunner_objects) for more information on how to use them.

## Variables

See also the how-to documentation on [variables](https://www.tensorflow.org/programmers_guide/variables) and [variable scopes](https://www.tensorflow.org/programmers_guide/variable_scope), and [the API documentation for variables](https://www.tensorflow.org/api_guides/python/state_ops).

#### What is the lifetime of a variable?

A variable is created when you first run the [tf.Variable.initializer](https://www.tensorflow.org/api_docs/python/tf/Variable#initializer) operation for that variable in a session. It is destroyed when that [tf.Session.close](https://www.tensorflow.org/api_docs/python/tf/Session#close).

#### How do variables behave when they are concurrently accessed?

Variables allow concurrent read and write operations. The value read from a variable may change if it is concurrently updated. By default, concurrent assigment operations to a variable are allowed to run with no mutual exclusion. To acquire a lock when assigning to a variable, pass use\_locking=True to [tf.Variable.assign](https://www.tensorflow.org/api_docs/python/tf/Variable#assign).

## Tensor shapes

See also the [tf.TensorShape](https://www.tensorflow.org/api_docs/python/tf/TensorShape).

#### How can I determine the shape of a tensor in Python?

In TensorFlow, a tensor has both a static (inferred) shape and a dynamic (true) shape. The static shape can be read using the [tf.Tensor.get\_shape](https://www.tensorflow.org/api_docs/python/tf/Tensor#get_shape) method: this shape is inferred from the operations that were used to create the tensor, and may be [partially complete](https://www.tensorflow.org/api_docs/python/tf/TensorShape). If the static shape is not fully defined, the dynamic shape of a Tensor t can be determined by evaluating [tf.shape(t)](https://www.tensorflow.org/api_docs/python/tf/shape).

#### What is the difference between x.set\_shape() and x = tf.reshape(x)?

The [tf.Tensor.set\_shape](https://www.tensorflow.org/api_docs/python/tf/Tensor#set_shape) method updates the static shape of a Tensor object, and it is typically used to provide additional shape information when this cannot be inferred directly. It does not change the dynamic shape of the tensor.

The [tf.reshape](https://www.tensorflow.org/api_docs/python/tf/reshape) operation creates a new tensor with a different dynamic shape.

#### How do I build a graph that works with variable batch sizes?

It is often useful to build a graph that works with variable batch sizes, for example so that the same code can be used for (mini-)batch training, and single-instance inference. The resulting graph can be [saved as a protocol buffer](https://www.tensorflow.org/api_docs/python/tf/Graph#as_graph_def) and [imported into another program](https://www.tensorflow.org/api_docs/python/tf/import_graph_def).

When building a variable-size graph, the most important thing to remember is not to encode the batch size as a Python constant, but instead to use a symbolic Tensor to represent it. The following tips may be useful:

* Use [batch\_size = tf.shape(input)[0]](https://www.tensorflow.org/api_docs/python/array_ops#shape) to extract the batch dimension from a Tensor called input, and store it in a Tensor called batch\_size.
* Use [tf.reduce\_mean](https://www.tensorflow.org/api_docs/python/tf/reduce_mean) instead of tf.reduce\_sum(...) / batch\_size.
* If you use [placeholders for feeding input](https://www.tensorflow.org/programmers_guide/reading_data#feeding), you can specify a variable batch dimension by creating the placeholder with [tf.placeholder(..., shape=[None, ...])](https://www.tensorflow.org/api_docs/python/io_ops#placeholder). The None element of the shape corresponds to a variable-sized dimension.

## TensorBoard

#### How can I visualize a TensorFlow graph?

See the [graph visualization tutorial](https://www.tensorflow.org/get_started/graph_viz).

#### What is the simplest way to send data to TensorBoard?

Add summary ops to your TensorFlow graph, and write these summaries to a log directory. Then, start TensorBoard using

python tensorflow/tensorboard/tensorboard.py --logdir=path/to/log-directory

For more details, see the [Summaries and TensorBoard tutorial](https://www.tensorflow.org/get_started/summaries_and_tensorboard).

#### Every time I launch TensorBoard, I get a network security popup!

You can change TensorBoard to serve on localhost rather than '0.0.0.0' by the flag --host=localhost. This should quiet any security warnings.

## Extending TensorFlow

See also the how-to documentation for [adding a new operation to TensorFlow](https://www.tensorflow.org/extend/adding_an_op).

#### My data is in a custom format. How do I read it using TensorFlow?

There are two main options for dealing with data in a custom format.

The easier option is to write parsing code in Python that transforms the data into a numpy array, then feed a[tf.placeholder](https://www.tensorflow.org/api_docs/python/tf/placeholder) a tensor with that data. See the documentation on [using placeholders for input](https://www.tensorflow.org/programmers_guide/reading_data#feeding) for more details. This approach is easy to get up and running, but the parsing can be a performance bottleneck.

The more efficient option is to [add a new op written in C++](https://www.tensorflow.org/extend/adding_an_op) that parses your data format. The [guide to handling new data formats](https://www.tensorflow.org/extend/new_data_formats) has more information about the steps for doing this.

#### How do I define an operation that takes a variable number of inputs?

The TensorFlow op registration mechanism allows you to define inputs that are a single tensor, a list of tensors with the same type (for example when adding together a variable-length list of tensors), or a list of tensors with different types (for example when enqueuing a tuple of tensors to a queue). See the how-to documentation for[adding an op with a list of inputs or outputs](https://www.tensorflow.org/extend/adding_an_op#list_inputs_and_outputs) for more details of how to define these different input types.

## Miscellaneous

#### What is TensorFlow's coding style convention?

The TensorFlow Python API adheres to the [PEP8](https://www.python.org/dev/peps/pep-0008/) conventions.\* In particular, we use CamelCase names for classes, and snake\_case names for functions, methods, and properties. We also adhere to the [Google Python style guide](https://google.github.io/styleguide/pyguide.html).

The TensorFlow C++ code base adheres to the [Google C++ style guide](http://google.github.io/styleguide/cppguide.html).

(\* With one exception: we use 2-space indentation instead of 4-space indentation.)