TensorFlow 2.0 Beta is available **Learn more** (/beta/)

Variables

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

Variables are manipulated via the `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) class. A `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) represents a tensor whose value can be changed by running ops on it. Unlike `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects, a `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) exists outside the context of a single `session.run` call.

Internally, a `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) stores a persistent tensor. Specific ops allow you to read and modify the values of this tensor. These modifications are visible across multiple `tf.Session` (https://www.tensorflow.org/api_docs/python/tf/Session)s, so multiple workers can see the same values for a `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable).

Creating a Variable

The best way to create a variable is to call the `tf.get_variable` (https://www.tensorflow.org/api_docs/python/tf/get_variable) function. This function requires you to specify the Variable's name. This name will be used by other replicas to access the same variable, as well as to name this variable's value when checkpointing and exporting models. `tf.get_variable` (https://www.tensorflow.org/api_docs/python/tf/get_variable) also allows you to reuse a previously created variable of the same name, making it easy to define models which reuse layers.

To create a variable with **`tf.get_variable`** (https://www.tensorflow.org/api_docs/python/tf/get_variable), simply provide the name and shape

```
my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

This creates a variable named "my_variable" which is a three-dimensional tensor with shape `[1, 2, 3]`. This variable will, by default, have the `dtype` **`tf.float32`** (https://www.tensorflow.org/api_docs/python/tf#float32) and its initial value will be randomized via **`tf.glorot_uniform_initializer`** (https://www.tensorflow.org/api_docs/python/tf/glorot_uniform_initializer).

You may optionally specify the `dtype` and initializer to **`tf.get_variable`** (https://www.tensorflow.org/api_docs/python/tf/get_variable). For example:

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3], dtype=tf.int32,
  initializer=tf.zeros_initializer)
```

TensorFlow provides many convenient initializers. Alternatively, you may initialize a **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) to have the value of a **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor). For example:

```
other_variable = tf.get_variable("other_variable", dtype=tf.int32,
  initializer=tf.constant([23, 42]))
```

Note that when the initializer is a **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor) you should not specify the variable's shape, as the shape of the initializer tensor will be used.

Variable collections

Because disconnected parts of a TensorFlow program might want to create variables, it is sometimes useful to have a single way to access all of them. For this reason TensorFlow provides **collections**, which are named lists of tensors or other objects, such as **tf.Variable** (https://www.tensorflow.org/api_docs/python/tf/Variable) instances.

By default every **tf.Variable** (https://www.tensorflow.org/api_docs/python/tf/Variable) gets placed in the following two collections:

- **tf.GraphKeys.GLOBAL_VARIABLES** (https://www.tensorflow.org/api_docs/python/tf/GraphKeys#GLOBAL_VARIABLES) --- variables that can be shared across multiple devices,
- **tf.GraphKeys.TRAINABLE_VARIABLES** (https://www.tensorflow.org/api_docs/python/tf/GraphKeys#TRAINABLE_VARIABLES) --- variables for which TensorFlow will calculate gradients.

If you don't want a variable to be trainable, add it to the **tf.GraphKeys.LOCAL_VARIABLES** (https://www.tensorflow.org/api_docs/python/tf/GraphKeys#LOCAL_VARIABLES) collection instead. For example, the following snippet demonstrates how to add a variable named `my_local` to this collection:

```
my_local = tf.get_variable("my_local", shape=(),
collections=[tf.GraphKeys.LOCAL_VARIABLES])
```

Alternatively, you can specify `trainable=False` as an argument to **tf.get_variable** (https://www.tensorflow.org/api_docs/python/tf/get_variable):

```
my_non_trainable = tf.get_variable("my_non_trainable",
                                   shape=(),
                                   trainable=False)
```

You can also use your own collections. Any string is a valid collection name, and there is no need to explicitly create a collection. To add a variable (or any other object) to a collection after creating the variable, call **tf.add_to_collection** (https://www.tensorflow.org/api_docs/python/tf/add_to_collection). For example, the following code adds an existing variable named `my_local` to a collection named `my_collection_name`:

```
tf.add_to_collection("my_collection_name", my_local)
```

And to retrieve a list of all the variables (or other objects) you've placed in a collection you can use:

```
tf.get_collection("my_collection_name")
```

Device placement

Just like any other TensorFlow operation, you can place variables on particular devices. For example, the following snippet creates a variable named v and places it on the second GPU device:

```
with tf.device("/device:GPU:1"):
  v = tf.get_variable("v", [1])
```

It is particularly important for variables to be in the correct device in distributed settings. Accidentally putting variables on workers instead of parameter servers, for example, can severely slow down training or, in the worst case, let each worker blithely forge ahead with its own independent copy of each variable. For this reason we provide **`tf.train.replica_device_setter`** (https://www.tensorflow.org/api_docs/python/tf/train/replica_device_setter), which can automatically place variables in parameter servers. For example:

```
cluster_spec = {
    "ps": ["ps0:2222", "ps1:2222"],
    "worker": ["worker0:2222", "worker1:2222", "worker2:2222"]}
with tf.device(tf.train.replica_device_setter(cluster=cluster_spec)):
  v = tf.get_variable("v", shape=[20, 20])  # this variable is placed
                                            # in the parameter server
                                            # by the replica_device_setter
```

Initializing variables

Before you can use a variable, it must be initialized. If you are programming in the low-level TensorFlow API (that is, you are explicitly creating your own graphs and sessions), you must explicitly initialize the variables. Most high-level frameworks such as **`tf.contrib.slim`** (https://www.tensorflow.org/api_docs/python/tf/contrib/slim), **`tf.estimator.Estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) and `Keras` automatically initialize variables for you before training a model.

Explicit initialization is otherwise useful because it allows you not to rerun potentially expensive initializers when reloading a model from a checkpoint as well as allowing determinism when randomly-initialized variables are shared in a distributed setting.

To initialize all trainable variables in one go, before training starts, call **`tf.global_variables_initializer()`** (https://www.tensorflow.org/api_docs/python/tf/initializers/global_variables). This function returns a single operation responsible for initializing all variables in the **`tf.GraphKeys.GLOBAL_VARIABLES`** (https://www.tensorflow.org/api_docs/python/tf/GraphKeys#GLOBAL_VARIABLES) collection. Running this operation initializes all variables. For example:

```
session.run(tf.global_variables_initializer())
# Now all variables are initialized.
```

If you do need to initialize variables yourself, you can run the variable's initializer operation. For example:

```
session.run(my_variable.initializer)
```

You can also ask which variables have still not been initialized. For example, the following code prints the names of all variables which have not yet been initialized:

```
print(session.run(tf.report_uninitialized_variables()))
```

Note that by default **`tf.global_variables_initializer`** (https://www.tensorflow.org/api_docs/python/tf/initializers/global_variables) does not specify the order in which variables are initialized. Therefore, if the initial value of a variable depends on another variable's value, it's likely that you'll get an error. Any time you use the value of a variable in a context in which not all variables are initialized (say, if you use a variable's value while initializing another variable), it is best to use `variable.initialized_value()` instead of `variable`:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
w = tf.get_variable("w", initializer=v.initialized_value() + 1)
```

Using variables

To use the value of a **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) in a TensorFlow graph, simply treat it like a normal **`tf.Tensor`** (https://www.tensorflow.org/api_docs/python/tf/Tensor):

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
w = v + 1  # w is a tf.Tensor which is computed based on the value of v.
           # Any time a variable is used in an expression it gets automatically
           # converted to a tf.Tensor representing its value.
```

To assign a value to a variable, use the methods `assign`, `assign_add`, and friends in the **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) class. For example, here is how you can call these methods:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
assignment = v.assign_add(1)
tf.global_variables_initializer().run()
sess.run(assignment)  # or assignment.op.run(), or assignment.eval()
```

Most TensorFlow optimizers have specialized ops that efficiently update the values of variables according to some gradient descent-like algorithm. See

`tf.train.Optimizer` (https://www.tensorflow.org/api_docs/python/tf/train/Optimizer) for an explanation of how to use optimizers.

Because variables are mutable it's sometimes useful to know what version of a variable's value is being used at any point in time. To force a re-read of the value of a variable after something has happened, you can use **`tf.Variable.read_value`** (https://www.tensorflow.org/api_docs/python/tf/Variable#read_value). For example:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
assignment = v.assign_add(1)
with tf.control_dependencies([assignment]):
  w = v.read_value()  # w is guaranteed to reflect v's value after the
                      # assign_add operation.
```

Sharing variables

TensorFlow supports two ways of sharing variables:

- Explicitly passing **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) objects around.
- Implicitly wrapping **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) objects within **`tf.variable_scope`** (https://www.tensorflow.org/api_docs/python/tf/variable_scope) objects.

While code which explicitly passes variables around is very clear, it is sometimes convenient to write TensorFlow functions that implicitly use variables in their implementations. Most of the functional layers from **`tf.layers`** (https://www.tensorflow.org/api_docs/python/tf/layers) use this approach, as well as all **`tf.metrics`** (https://www.tensorflow.org/api_docs/python/tf/metrics), and a few other library utilities.

Variable scopes allow you to control variable reuse when calling functions which implicitly create and use variables. They also allow you to name your variables in a hierarchical and understandable way.

For example, let's say we write a function to create a convolutional / relu layer:

```
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`, which is good for clarity. In a real model, however, we want many such convolutional layers, and calling this function repeatedly would not work:

```
input1 = tf.random_normal([1,10,10,32])
input2 = tf.random_normal([1,20,20,32])
x = conv_relu(input1, kernel_shape=[5, 5, 32, 32], bias_shape=[32])
x = conv_relu(x, kernel_shape=[5, 5, 32, 32], bias_shape = [32])  # This fails.
```

Since the desired behavior is unclear (create new variables or reuse the existing ones?) TensorFlow will fail. Calling `conv_relu` in different scopes, however, clarifies that we want to create new 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])
```

If you do want the variables to be shared, you have two options. First, you can create a scope with the same name using `reuse=True`:

```
with tf.variable_scope("model"):
  output1 = my_image_filter(input1)
with tf.variable_scope("model", reuse=True):
  output2 = my_image_filter(input2)
```

You can also call `scope.reuse_variables()` to trigger a reuse:

```
with tf.variable_scope("model") as scope:
  output1 = my_image_filter(input1)
  scope.reuse_variables()
  output2 = my_image_filter(input2)
```

Since depending on exact string names of scopes can feel dangerous, it's also possible to initialize a variable scope based on another one:

```
with tf.variable_scope("model") as scope:
  output1 = my_image_filter(input1)
with tf.variable_scope(scope, reuse=True):
  output2 = my_image_filter(input2)
```