Constants, Sequences, and Random Values

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

Contents

- `tf.random_crop(value, size, seed=None, name=None)`
- `tf.multinomial(logits, num_samples, seed=None, name=None)`
- `tf.random_gamma(shape, alpha, beta=None, dtype=tf.float32, seed=None, name=None)`
- `tf.set_random_seed(seed)`

Constant Value Tensors

TensorFlow provides several operations that you can use to generate constants.

`tf.zeros(shape, dtype=tf.float32, name=None)`

Creates a tensor with all elements set to zero.

This operation returns a tensor of type `dtype` with shape `shape` and all elements set to zero.

For example:

```
tf.zeros([3, 4], tf.int32) ==> [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Args:

- `shape`: Either a list of integers, or a 1-D `Tensor` of type `int32`.
- `dtype`: The type of an element in the resulting `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` with all elements set to zero.

tf.zeros_like(tensor, dtype=None, name=None, optimize=True)

Creates a tensor with all elements set to zero.

Given a single tensor (`tensor`), this operation returns a tensor of the same type and shape as `tensor` with all elements set to zero. Optionally, you can use `dtype` to specify a new type for the returned tensor.

For example:

```
# 'tensor' is [[1, 2, 3], [4, 5, 6]]
tf.zeros_like(tensor) ==> [[0, 0, 0], [0, 0, 0]]
```

Args:

- `tensor`: A `Tensor`.

  `dtype`: A type for the returned `Tensor`. Must be `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `complex64`, or `complex128`.

  `name`: A name for the operation (optional).

  `optimize`: if true, attempt to statically determine the shape of 'tensor' and encode it as a constant.

Returns:

A `Tensor` with all elements set to zero.

---

tf.ones(shape, dtype=tf.float32, name=None)

Creates a tensor with all elements set to 1.

This operation returns a tensor of type `dtype` with shape `shape` and all elements set to 1.

For example:

```
tf.ones([2, 3], tf.int32) ==> [[1, 1, 1], [1, 1, 1]]
```

Args:

- `shape`: Either a list of integers, or a 1-D `Tensor` of type `int32`.
- `dtype`: The type of an element in the resulting `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` with all elements set to 1.

`tf.ones_like(tensor, dtype=None, name=None, optimize=True)`

Creates a tensor with all elements set to 1.

Given a single tensor (`tensor`), this operation returns a tensor of the same type and shape as `tensor` with all elements set to 1. Optionally, you can specify a new type (`dtype`) for the returned tensor.

For example:

```
# 'tensor' is [[1, 2, 3], [4, 5, 6]]
tf.ones_like(tensor) ==> [[1, 1, 1], [1, 1, 1]]
```

Args:

- `tensor`: A `Tensor`.
- `dtype`: A type for the returned `Tensor`. Must be `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `complex64`, `complex128` or `bool`.
- `name`: A name for the operation (optional).
- `optimize`: if true, attempt to statically determine the shape of 'tensor' and encode it as a constant.

Returns:

A `Tensor` with all elements set to 1.

`tf.fill(dims, value, name=None)`

Creates a tensor filled with a scalar value.

This operation creates a tensor of shape `dims` and fills it with `value`.

For example:

```
# Output tensor has shape [2, 3].
fill([2, 3], 9) ==> [[9, 9, 9]
                     [9, 9, 9]]
```

Args:

- `dims`: A `Tensor` of type `int32`. 1-D. Represents the shape of the output tensor.
- `value`: A `Tensor`. 0-D (scalar). Value to fill the returned tensor.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `value`.

`tf.constant(value, dtype=None, shape=None, name='Const')`

Creates a constant tensor.

The resulting tensor is populated with values of type `dtype`, as specified by arguments `value` and (optionally) `shape` (see examples below).

The argument `value` can be a constant value, or a list of values of type `dtype`. If `value` is a list, then the length of the list must be less than or equal to the number of elements implied by the `shape` argument (if specified). In the case where the list length is less than the number of elements specified by `shape`, the last element in the list will be used to fill the remaining entries.

The argument `shape` is optional. If present, it specifies the dimensions of the resulting tensor. If not present, the shape of `value` is used.

If the argument `dtype` is not specified, then the type is inferred from the type of `value`.

For example:

```
# Constant 1-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3, 4, 5, 6, 7]) => [1 2 3 4 5 6 7]

# Constant 2-D tensor populated with scalar value -1.
tensor = tf.constant(-1.0, shape=[2, 3]) => [[-1. -1. -1.]
                                             [-1. -1. -1.]]
```

Args:

`value`: A constant value (or list) of output type `dtype`.

`dtype`: The type of the elements of the resulting tensor.

`shape`: Optional dimensions of resulting tensor.

`name`: Optional name for the tensor.

Returns:

A Constant Tensor.

Sequences

`tf.linspace(start, stop, num, name=None)`

Generates values in an interval.

A sequence of `num` evenly-spaced values are generated beginning at `start`. If `num > 1`, the values in the sequence increase by `stop - start / num - 1`, so that the last one is exactly `stop`.

For example:

```
tf.linspace(10.0, 12.0, 3, name="linspace") => [ 10.0  11.0  12.0]
```

Args:

- `start`: A `Tensor`. Must be one of the following types: `float32`, `float64`. First entry in the range.
- `stop`: A `Tensor`. Must have the same type as `start`. Last entry in the range.
- `num`: A `Tensor`. Must be one of the following types: `int32`, `int64`. Number of values to generate.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `start`. 1-D. The generated values.

`tf.range(start, limit=None, delta=1, name='range')`

Creates a sequence of integers.

Creates a sequence of integers that begins at `start` and extends by increments of `delta` up to but not including `limit`.

Like the Python builtin `range`, `start` defaults to 0, so that `range(n) = range(0, n)`.

For example:

```
# 'start' is 3
# 'limit' is 18
# 'delta' is 3
tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]

# 'limit' is 5
tf.range(limit) ==> [0, 1, 2, 3, 4]
```

Args:

- `start`: A 0-D (scalar) of type `int32`. Acts as first entry in the range if `limit` is not None; otherwise, acts as range limit and first entry defaults to 0.
- `limit`: A 0-D (scalar) of type `int32`. Upper limit of sequence, exclusive. If None, defaults to the value of `start` while the first entry of the range defaults to 0.
- `delta`: A 0-D `Tensor` (scalar) of type `int32`. Number that increments `start`. Defaults to 1.
- `name`: A name for the operation. Defaults to "range".

Returns:

An 1-D `int32` `Tensor`.

Random Tensors

TensorFlow has several ops that create random tensors with different distributions. The random ops are stateful, and create new random values each time they are evaluated.

The `seed` keyword argument in these functions acts in conjunction with the graph-level random seed. Changing either the graph-level seed using `set_random_seed` or the op-level seed will change the underlying seed of these operations. Setting neither graph-level nor op-level seed, results in a random seed for all operations. See `set_random_seed` for details on the interaction between operation-level and graph-level random seeds.

Examples:

```
# Create a tensor of shape [2, 3] consisting of random normal values, with mean
# -1 and standard deviation 4.
norm = tf.random_normal([2, 3], mean=-1, stddev=4)

# Shuffle the first dimension of a tensor
c = tf.constant([[1, 2], [3, 4], [5, 6]])
shuff = tf.random_shuffle(c)

# Each time we run these ops, different results are generated
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))

# Set an op-level seed to generate repeatable sequences across sessions.
norm = tf.random_normal([2, 3], seed=1234)
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))
```

Another common use of random values is the initialization of variables. Also see the Variables How To.

```
# Use random uniform values in [0, 1) as the initializer for a variable of shape
# [2, 3]. The default type is float32.
var = tf.Variable(tf.random_uniform([2, 3]), name="var")
init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)
print(sess.run(var))
```

`tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)`

Outputs random values from a normal distribution.

Args:

- `shape`: A 1-D integer Tensor or Python array. The shape of the output tensor.
- `mean`: A 0-D Tensor or Python value of type `dtype`. The mean of the normal distribution.
- `stddev`: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution.
- `dtype`: The type of the output.
- `seed`: A Python integer. Used to create a random seed for the distribution. See `set_random_seed` for behavior.
- `name`: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random normal values.

`tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)`

Outputs random values from a truncated normal distribution.

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

Args:

- `shape`: A 1-D integer Tensor or Python array. The shape of the output tensor.
- `mean`: A 0-D Tensor or Python value of type `dtype`. The mean of the truncated normal distribution.
- `stddev`: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the truncated normal distribution.
- `dtype`: The type of the output.
- `seed`: A Python integer. Used to create a random seed for the distribution. See `set_random_seed` for behavior.
- `name`: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random truncated normal values.

`tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)`

Outputs random values from a uniform distribution.

The generated values follow a uniform distribution in the range `[minval, maxval)`. The lower bound `minval` is included in the range, while the upper bound `maxval` is excluded.

For floats, the default range is `[0, 1)`. For ints, at least `maxval` must be specified explicitly.

In the integer case, the random integers are slightly biased unless `maxval - minval` is an exact power of two. The bias is small for values of `maxval - minval` significantly smaller than the range of the output (either `2**32` or `2**64`).

Args:

- `shape`: A 1-D integer Tensor or Python array. The shape of the output tensor.
- `minval`: A 0-D Tensor or Python value of type `dtype`. The lower bound on the range of random values to generate. Defaults to 0.
- `maxval`: A 0-D Tensor or Python value of type `dtype`. The upper bound on the range of random values to generate. Defaults to 1 if `dtype` is floating point.
- `dtype`: The type of the output: `float32`, `float64`, `int32`, or `int64`.
- `seed`: A Python integer. Used to create a random seed for the distribution. See `set_random_seed` for behavior.
- `name`: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random uniform values.

Raises:

- `ValueError`: If `dtype` is integral and `maxval` is not specified.

`tf.random_shuffle(value, seed=None, name=None)`

Randomly shuffles a tensor along its first dimension.

The tensor is shuffled along dimension 0, such that each `value[j]` is mapped to one and only one `output[i]`. For example, a mapping that might occur for a 3x2 tensor is:

```
[[1, 2],       [[5, 6],
 [3, 4],  ==>   [1, 2],
 [5, 6]]        [3, 4]]
```

Args:

- `value`: A Tensor to be shuffled.
- `seed`: A Python integer. Used to create a random seed for the distribution. See `set_random_seed` for behavior.
- `name`: A name for the operation (optional).

Returns:

A tensor of same shape and type as `value`, shuffled along its first dimension.

`tf.random_crop(value, size, seed=None, name=None)`

Randomly crops a tensor to a given size.

Slices a shape `size` portion out of `value` at a uniformly chosen offset. Requires `value.shape >= size`.

If a dimension should not be cropped, pass the full size of that dimension. For example, RGB images can be cropped with `size = [crop_height, crop_width, 3]`.

Args:

- `value`: Input tensor to crop.
- `size`: 1-D tensor with size the rank of `value`.

- `seed`: Python integer. Used to create a random seed. See `set_random_seed` for behavior.
- `name`: A name for this operation (optional).

Returns:

A cropped tensor of the same rank as `value` and shape `size`.

`tf.multinomial(logits, num_samples, seed=None, name=None)`

Draws samples from a multinomial distribution.

Example:

```
# samples has shape [1, 5], where each value is either 0 or 1 with equal
# probability.
samples = tf.multinomial(tf.log([[10., 10.]]), 5)
```

Args:

- `logits`: 2-D Tensor with shape `[batch_size, num_classes]`. Each slice `[i, :]` represents the unnormalized log probabilities for all classes.
- `num_samples`: 0-D. Number of independent samples to draw for each row slice.
- `seed`: A Python integer. Used to create a random seed for the distribution. See `set_random_seed` for behavior.
- `name`: Optional name for the operation.

Returns:

The drawn samples of shape `[batch_size, num_samples]`.

`tf.random_gamma(shape, alpha, beta=None, dtype=tf.float32, seed=None, name=None)`

Draws `shape` samples from each of the given Gamma distribution(s).

`alpha` is the shape parameter describing the distribution(s), and `beta` is the inverse scale parameter(s).

Example:

samples = tf.random_gamma([10], [0.5, 1.5]) # samples has shape [10, 2], where each slice [:, 0] and [:, 1] represents # the samples drawn from each distribution

samples = tf.random_gamma([7, 5], [0.5, 1.5]) # samples has shape [7, 5, 2], where each slice [:, :, 0] and [:, :, 1] # represents the 7x5 samples drawn from each of the two distributions

samples = tf.random_gamma([30], [[1.],[3.],[5.]], beta=[[3., 4.]]) # samples has shape [30, 3, 2], with 30 samples each of 3x2 distributions.

Note that for small alpha values, there is a chance you will draw a value of exactly 0, which gets worse for lower-precision dtypes, even though zero is not in the support of the gamma distribution.

Relevant cdfs (~chance you will draw a exactly-0 value):
`stats.gamma(.01).cdf(np.finfo(np.float16).tiny) 0.91269738769897879`
`stats.gamma(.01).cdf(np.finfo(np.float32).tiny) 0.41992668622045726`
`stats.gamma(.01).cdf(np.finfo(np.float64).tiny) 0.00084322740680686662`
`stats.gamma(.35).cdf(np.finfo(np.float16).tiny) 0.037583276135263931`
`stats.gamma(.35).cdf(np.finfo(np.float32).tiny) 5.9514895726818067e-14`
`stats.gamma(.35).cdf(np.finfo(np.float64).tiny) 2.3529843400647272e-108`

Args:

- `shape`: A 1-D integer Tensor or Python array. The shape of the output samples to be drawn per alpha/beta-parameterized distribution.
- `alpha`: A Tensor or Python value or N-D array of type `dtype`. `alpha` provides the shape parameter(s) describing the gamma distribution(s) to sample. Must be broadcastable with `beta`.

- `beta`: A Tensor or Python value or N-D array of type `dtype`. Defaults to 1. `beta` provides the inverse scale parameter(s) of the gamma distribution(s) to sample. Must be broadcastable with `alpha`.

- `dtype`: The type of alpha, beta, and the output: `float16`, `float32`, or `float64`.

- `seed`: A Python integer. Used to create a random seed for the distributions. See `set_random_seed` for behavior.

- `name`: Optional name for the operation.

Returns:

- `samples`: a `Tensor` of shape `tf.concat(shape, tf.shape(alpha + beta))` with values of type `dtype`.

`tf.set_random_seed(seed)`

Sets the graph-level random seed.

Operations that rely on a random seed actually derive it from two seeds: the graph-level and operation-level seeds. This sets the graph-level seed.

Its interactions with operation-level seeds is as follows:

1. If neither the graph-level nor the operation seed is set: A random seed is used for this op.
2. If the graph-level seed is set, but the operation seed is not: The system deterministically picks an operation seed in conjunction with the graph-level seed so that it gets a unique random sequence.
3. If the graph-level seed is not set, but the operation seed is set: A default graph-level seed and the specified operation seed are used to determine the random sequence.
4. If both the graph-level and the operation seed are set: Both seeds are used in conjunction to determine the random sequence.

To illustrate the user-visible effects, consider these examples:

To generate different sequences across sessions, set neither graph-level nor op-level seeds:

```
a = tf.random_uniform([1])
b = tf.random_normal([1])

print("Session 1")
with tf.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
  print(sess2.run(a))  # generates 'A3'
  print(sess2.run(a))  # generates 'A4'
  print(sess2.run(b))  # generates 'B3'
  print(sess2.run(b))  # generates 'B4'
```

To generate the same repeatable sequence for an op across sessions, set the seed for the op:

```
a = tf.random_uniform([1], seed=1)
b = tf.random_normal([1])

# Repeatedly running this block with the same graph will generate the same
# sequence of values for 'a', but different sequences of values for 'b'.
print("Session 1")
with tf.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
  print(sess2.run(a))  # generates 'A1'
  print(sess2.run(a))  # generates 'A2'
  print(sess2.run(b))  # generates 'B3'
  print(sess2.run(b))  # generates 'B4'
```

To make the random sequences generated by all ops be repeatable across sessions, set a graph-level seed:

```
tf.set_random_seed(1234)
a = tf.random_uniform([1])
b = tf.random_normal([1])
```

```
# Repeatedly running this block with the same graph will generate different
# sequences of 'a' and 'b'.
print("Session 1")
with tf.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
  print(sess2.run(a))  # generates 'A1'
  print(sess2.run(a))  # generates 'A2'
  print(sess2.run(b))  # generates 'B1'
  print(sess2.run(b))  # generates 'B2'
```

Args:

- `seed`: integer.