目录

Building Graphs

Contents

- `class tf.bytes`

Classes and functions for building TensorFlow graphs.

Core graph data structures

`class tf.Graph`

A TensorFlow computation, represented as a dataflow graph.

A `Graph` contains a set of `Operation` objects, which represent units of computation; and `Tensor` objects, which represent the units of data that flow between operations.

A default `Graph` is always registered, and accessible by calling `tf.get_default_graph()`. To add an operation to the default graph, simply call one of the functions that defines a new `Operation`:

```
c = tf.constant(4.0)
assert c.graph is tf.get_default_graph()
```

Another typical usage involves the `Graph.as_default()` context manager, which overrides the current default graph for the lifetime of the context:

```
g = tf.Graph()
with g.as_default():
  # Define operations and tensors in `g`.
  c = tf.constant(30.0)
  assert c.graph is g
```

Important note: This class is not thread-safe for graph construction. All operations should be created from a single thread, or external synchronization must be provided. Unless otherwise specified, all methods are not thread-safe.

`tf.Graph.__init__()`

Creates a new, empty Graph.

`tf.Graph.as_default()`

Returns a context manager that makes this `Graph` the default graph.

This method should be used if you want to create multiple graphs in the same process. For convenience, a global default graph is provided, and all ops will be added to this graph if you do not create a new graph explicitly. Use this method the `with` keyword to specify that ops created within the scope of a block should be added to this graph.
The default graph is a property of the current thread. If you create a new thread, and wish to use the default graph in that thread, you must explicitly add a `with g.as_default():` in that thread's function.

The following code examples are equivalent:

```
# 1. Using Graph.as_default():
g = tf.Graph()
with g.as_default():
  c = tf.constant(5.0)
  assert c.graph is g

# 2. Constructing and making default:
with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  assert c.graph is g
```

Returns:

A context manager for using this graph as the default graph.

`tf.Graph.as_graph_def(from_version=None, add_shapes=False)`

Returns a serialized `GraphDef` representation of this graph.

The serialized `GraphDef` can be imported into

another `Graph` (using `import_graph_def()`) or used with the C++ Session API.

This method is thread-safe.

Args:

- `from_version`: Optional. If this is set, returns a `GraphDef` containing only the nodes that were added to this graph since its `version` property had the given value.
- `add_shapes`: If true, adds an "_output_shapes" list attr to each node with the inferred shapes of each of its outputs.

Returns:

A `GraphDef` protocol buffer.

Raises:

- `ValueError`: If the `graph_def` would be too large.

`tf.Graph.finalize()`

Finalizes this graph, making it read-only.

After calling `g.finalize()`, no new operations can be added to `g`. This method is used to ensure that no operations are added to a graph when it is shared between multiple threads, for example when using a`QueueRunner`.

`tf.Graph.finalized`

True if this graph has been finalized.

`tf.Graph.control_dependencies(control_inputs)`

Returns a context manager that specifies control dependencies.

Use with the `with` keyword to specify that all operations constructed within the context should have control dependencies on `control_inputs`. For example:

```
with g.control_dependencies([a, b, c]):
  # `d` and `e` will only run after `a`, `b`, and `c` have
executed.
  d = ...
  e = ...
```

Multiple calls to `control_dependencies()` can be nested, and in that case a new `Operation` will have control dependencies on the union of `control_inputs` from all active contexts.

```
with g.control_dependencies([a, b]):
  # Ops constructed here run after `a` and `b`.
  with g.control_dependencies([c, d]):
    # Ops constructed here run after `a`, `b`, `c`, and `d`.
```

You can pass None to clear the control dependencies:

```
with g.control_dependencies([a, b]):
  # Ops constructed here run after `a` and `b`.
  with g.control_dependencies(None):
    # Ops constructed here run normally, not waiting for either
`a` or `b`.
    with g.control_dependencies([c, d]):
      # Ops constructed here run after `c` and `d`, also not
waiting
      # for either `a` or `b`.
```

N.B. The control dependencies context applies only to ops that are constructed within the context. Merely using an op or tensor in the context does not add a control dependency. The following example illustrates this point:

```
# WRONG
def my_func(pred, tensor):
  t = tf.matmul(tensor, tensor)
  with tf.control_dependencies([pred]):
    # The matmul op is created outside the context, so no control
    # dependency will be added.
    return t

# RIGHT
def my_func(pred, tensor):
  with tf.control_dependencies([pred]):
    # The matmul op is created in the context, so a control
dependency
    # will be added.
    return tf.matmul(tensor, tensor)
```

Args:

- `control_inputs`: A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies.

Returns:

A context manager that specifies control dependencies for all operations constructed within the context.

Raises:

- `TypeError`: If `control_inputs` is not a list of `Operation` or `Tensor` objects.

`tf.Graph.device(device_name_or_function)`

Returns a context manager that specifies the default device to use.

The `device_name_or_function` argument may either be a device name string, a device function, or None:

- If it is a device name string, all operations constructed in this context will be assigned to the device with that name, unless overridden by a nested `device()` context.
- If it is a function, it will be treated as function from Operation objects to device name strings, and invoked each time a new Operation is created. The Operation will be assigned to the device with the returned name.
- If it is None, all `device()` invocations from the enclosing context will be ignored.

For example:

```
with g.device('/gpu:0'):
  # All operations constructed in this context will be placed
  # on GPU 0.
  with g.device(None):
    # All operations constructed in this context will have no
    # assigned device.

# Defines a function from `Operation` to device string.
def matmul_on_gpu(n):
  if n.type == "MatMul":
    return "/gpu:0"
  else:
```

```
    return "/cpu:0"

with g.device(matmul_on_gpu):
  # All operations of type "MatMul" constructed in this context
  # will be placed on GPU 0; all other operations will be placed
  # on CPU 0.
```

Args:

- `device_name_or_function`: The device name or function to use in the context.

Returns:

A context manager that specifies the default device to use for newly created ops.

`tf.Graph.name_scope(name)`

Returns a context manager that creates hierarchical names for operations.

A graph maintains a stack of name scopes. A `with name_scope(...):` statement pushes a new name onto the stack for the lifetime of the context.

The `name` argument will be interpreted as follows:

- A string (not ending with '/') will create a new name scope, in which `name` is appended to the prefix of all operations created in the context. If `name` has been used before, it will be made unique by calling`self.unique_name(name)`.

- A scope previously captured from a `with g.name_scope(...) as scope:` statement will be treated as an "absolute" name scope, which makes it possible to re-enter existing scopes.
- A value of `None` or the empty string will reset the current name scope to the top-level (empty) name scope.

For example:

```
with tf.Graph().as_default() as g:
  c = tf.constant(5.0, name="c")
  assert c.op.name == "c"
  c_1 = tf.constant(6.0, name="c")
  assert c_1.op.name == "c_1"

  # Creates a scope called "nested"
  with g.name_scope("nested") as scope:
    nested_c = tf.constant(10.0, name="c")
    assert nested_c.op.name == "nested/c"

    # Creates a nested scope called "inner".
    with g.name_scope("inner"):
      nested_inner_c = tf.constant(20.0, name="c")
      assert nested_inner_c.op.name == "nested/inner/c"

    # Create a nested scope called "inner_1".
    with g.name_scope("inner"):
      nested_inner_1_c = tf.constant(30.0, name="c")
      assert nested_inner_1_c.op.name == "nested/inner_1/c"

      # Treats `scope` as an absolute name scope, and
      # switches to the "nested/" scope.
      with g.name_scope(scope):
        nested_d = tf.constant(40.0, name="d")
        assert nested_d.op.name == "nested/d"

        with g.name_scope(""):
          e = tf.constant(50.0, name="e")
          assert e.op.name == "e"
```

The name of the scope itself can be captured by `with g.name_scope(...) as scope:`, which stores the name of the scope in the variable `scope`. This value can be used to name an operation that represents the overall result of executing the ops in a scope. For example:

```
inputs = tf.constant(...)
with g.name_scope('my_layer') as scope:
  weights = tf.Variable(..., name="weights")
  biases = tf.Variable(..., name="biases")
  affine = tf.matmul(inputs, weights) + biases
  output = tf.nn.relu(affine, name=scope)
```

Args:

- `name`: A name for the scope.

Returns:

A context manager that installs `name` as a new name scope.

A `Graph` instance supports an arbitrary number of "collections" that are identified by name. For convenience when building a large graph, collections can store groups of related objects: for example, the `tf.Variable` uses a collection (named `tf.GraphKeys.VARIABLES`) for all variables that are created during the construction of a graph. The caller may define additional collections by specifying a new name.

`tf.Graph.add_to_collection(name, value)`

Stores `value` in the collection with the given `name`.

Note that collections are not sets, so it is possible to add a value to a collection several times.

Args:

- `name`: The key for the collection. The `GraphKeys` class contains many standard names for collections.
- `value`: The value to add to the collection.

`tf.Graph.get_collection(name, scope=None)`

Returns a list of values in the collection with the given `name`.

Args:

- `name`: The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
- `scope`: (Optional.) If supplied, the resulting list is filtered to include only items whose name begins with this string.

Returns:

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. The list contains the values in the order under which they were collected.

`tf.Graph.as_graph_element(obj, allow_tensor=True, allow_operation=True)`

Returns the object referred to by `obj`, as an `Operation` or `Tensor`.

This function validates that `obj` represents an element of this graph, and gives an informative error message if it is not.

This function is the canonical way to get/validate an object of one of the allowed types from an external argument reference in the Session API.

This method may be called concurrently from multiple threads.

Args:

- `obj`: A `Tensor`, an `Operation`, or the name of a tensor or operation.

  Can also be any object with an `_as_graph_element()` method that returns a value of one of these types.
- `allow_tensor`: If true, `obj` may refer to a `Tensor`.
- `allow_operation`: If true, `obj` may refer to an `Operation`.

Returns:

The `Tensor` or `Operation` in the Graph corresponding to `obj`.

Raises:

- `TypeError`: If `obj` is not a type we support attempting to convert to types.
- `ValueError`: If `obj` is of an appropriate type but invalid. For example, an invalid string.
- `KeyError`: If `obj` is not an object in the graph.

`tf.Graph.get_operation_by_name(name)`

Returns the `Operation` with the given `name`.

This method may be called concurrently from multiple threads.

Args:

- `name`: The name of the `Operation` to return.

Returns:

The `Operation` with the given `name`.

Raises:

- `TypeError`: If `name` is not a string.
- `KeyError`: If `name` does not correspond to an operation in this graph.

`tf.Graph.get_tensor_by_name(name)`

Returns the `Tensor` with the given `name`.

This method may be called concurrently from multiple threads.

Args:

- `name`: The name of the `Tensor` to return.

Returns:

The `Tensor` with the given `name`.

Raises:

- `TypeError`: If `name` is not a string.
- `KeyError`: If `name` does not correspond to a tensor in this graph.

`tf.Graph.get_operations()`

Return the list of operations in the graph.

You can modify the operations in place, but modifications to the list such as inserts/delete have no effect on the list of operations known to the graph.

This method may be called concurrently from multiple threads.

Returns:

A list of Operations.

`tf.Graph.seed`

`tf.Graph.unique_name(name)`

Return a unique operation name for `name`.

Note: You rarely need to call `unique_name()` directly. Most of the time you just need to create `with g.name_scope()` blocks to generate structured names.

`unique_name` is used to generate structured names, separated by `"/"`, to help identify operations when debugging a graph. Operation names are displayed in error messages reported by the TensorFlow runtime, and in various visualization tools such as TensorBoard.

Args:

- `name`: The name for an operation.

Returns:

A string to be passed to `create_op()` that will be used to name the operation being created.

`tf.Graph.version`

Returns a version number that increases as ops are added to the graph.

Note that this is unrelated to the GraphDef version.

`tf.Graph.graph_def_versions`

The GraphDef version information of this graph.

For details on the meaning of each version, see `GraphDef`.

Returns:

A `VersionDef`.

```
tf.Graph.create_op(op_type, inputs, dtypes,
input_types=None, name=None, attrs=None, op_def=None,
compute_shapes=True, compute_device=True)
```

Creates an `Operation` in this graph.

This is a low-level interface for creating an `Operation`. Most programs will not call this method directly, and instead use the Python op constructors, such as `tf.constant()`, which add ops to the default graph.

Args:

- `op_type`: The `Operation` type to create. This corresponds to the `OpDef.name` field for the proto that defines the operation.
- `inputs`: A list of `Tensor` objects that will be inputs to the `Operation`.
- `dtypes`: A list of `DType` objects that will be the types of the tensors that the operation produces.
- `input_types`: (Optional.) A list of `DType`s that will be the types of the tensors that the operation consumes. By default, uses the base `DType` of each input in `inputs`. Operations that expect reference-typed inputs must specify `input_types` explicitly.
- `name`: (Optional.) A string name for the operation. If not specified, a name is generated based on `op_type`.

* `attrs`: (Optional.) A dictionary where the key is the attribute name (a string) and the value is the respective`attr` attribute of the `NodeDef` proto that will represent the operation (an `AttrValue` proto).
* `op_def`: (Optional.) The `OpDef` proto that describes the `op_type` that the operation will have.
* `compute_shapes`: (Optional.) If True, shape inference will be performed to compute the shapes of the outputs.
* `compute_device`: (Optional.) If True, device functions will be executed to compute the device property of the Operation.

Raises:

* `TypeError`: if any of the inputs is not a `Tensor`.

Returns:

An `Operation` object.

`tf.Graph.gradient_override_map(op_type_map)`

EXPERIMENTAL: A context manager for overriding gradient functions.

This context manager can be used to override the gradient function that will be used for ops within the scope of the context.

For example:

```
@tf.RegisterGradient("CustomSquare")
```

```
def _custom_square_grad(op, inputs):
  # ...

with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  s_1 = tf.square(c)  # Uses the default gradient for tf.square.
  with g.gradient_override_map({"Square": "CustomSquare"}):
    s_2 = tf.square(s_2)  # Uses _custom_square_grad to compute
the
                        # gradient of s_2.
```

Args:

- `op_type_map`: A dictionary mapping op type strings to alternative op type strings.

Returns:

A context manager that sets the alternative op type to be used for one or more ops created in that context.

Raises:

- `TypeError`: If `op_type_map` is not a dictionary mapping strings to strings.

Other Methods

`tf.Graph.add_to_collections(names, value)`

Stores `value` in the collections given by `names`.

Note that collections are not sets, so it is possible to add a value to a collection several times. This function makes sure that duplicates in `names` are ignored, but it will not check for pre-existing membership of `value` in any of the collections in `names`.

Args:

- `names`: The keys for the collections to add to. The `GraphKeys` class contains many standard names for collections.
- `value`: The value to add to the collections.

`tf.Graph.get_all_collection_keys()`

Returns a list of collections used in this graph.

`class tf.Operation`

Represents a graph node that performs computation on tensors.

An `Operation` is a node in a TensorFlow `Graph` that takes zero or more `Tensor` objects as input, and produces zero or more `Tensor` objects as output. Objects of type `Operation` are created by calling a Python op constructor (such as `tf.matmul()`) or `Graph.create_op()`.

For example `c = tf.matmul(a, b)` creates an `Operation` of type "MatMul" that takes tensors `a` and `b` as input, and produces `c` as output.

After the graph has been launched in a session, an `Operation` can be executed by passing it to`Session.run()`. `op.run()` is a shortcut for calling `tf.get_default_session().run(op)`.

`tf.Operation.name`

The full name of this operation.

`tf.Operation.type`

The type of the op (e.g. `"MatMul"`).

`tf.Operation.inputs`

The list of `Tensor` objects representing the data inputs of this op.

`tf.Operation.control_inputs`

The `Operation` objects on which this op has a control dependency. Before this op is executed, TensorFlow will ensure that the operations in `self.control_inputs` have finished executing. This mechanism can be used to run ops sequentially for performance reasons, or to ensure that the side effects of an op are observed in the correct order.

Returns:

A list of `Operation` objects.

`tf.Operation.outputs`

The list of `Tensor` objects representing the outputs of this op.

`tf.Operation.device`

The name of the device to which this op has been assigned, if any.

Returns:

The string name of the device to which this op has been assigned, or an empty string if it has not been assigned to a device.

`tf.Operation.graph`

The `Graph` that contains this operation.

`tf.Operation.run(feed_dict=None, session=None)`

Runs this operation in a `Session`.

Calling this method will execute all preceding operations that produce the inputs needed for this operation.

N.B. Before invoking `Operation.run()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

Args:

- `feed_dict`: A dictionary that maps `Tensor` objects to feed values.

  See `Session.run()` for a description of the valid feed values.
- `session`: (Optional.) The `Session` to be used to run to this operation. If none, the default session will be used.

`tf.Operation.get_attr(name)`

Returns the value of the attr of this op with the given `name`.

Args:

- `name`: The name of the attr to fetch.

Returns:

The value of the attr, as a Python object.

Raises:

- `ValueError`: If this op does not have an attr with the given `name`.

`tf.Operation.traceback`

Returns the call stack from when this operation was constructed.

Other Methods

`tf.Operation.__init__(node_def, g, inputs=None, output_types=None, control_inputs=None, input_types=None, original_op=None, op_def=None)`

Creates an `Operation`.

NOTE: This constructor validates the name of the `Operation` (passed as `node_def.name`). Valid `Operation`names match the following regular expression:

```
[A-Za-z0-9.][A-Za-z0-9_.\-/]*
```

Args:

- `node_def`: `graph_pb2.NodeDef`. `NodeDef` for the `Operation`. Used for attributes of`graph_pb2.NodeDef`, typically `name`, `op`, and `device`.

  The `input` attribute is irrelevant here as it will be computed when generating the model.
- `g`: `Graph`. The parent graph.
- `inputs`: list of `Tensor` objects. The inputs to this `Operation`.
- `output_types`: list of `DType` objects. List of the types of the `Tensors` computed by this operation. The length of this list indicates the number of output endpoints of the `Operation`.
- `control_inputs`: list of operations or tensors from which to have a control dependency.

- `input_types`: List of `DType` objects representing the types of the tensors accepted by the `Operation`. By default uses `[x.dtype.base_dtype for x in inputs]`. Operations that expect reference-typed inputs must specify these explicitly.
- `original_op`: Optional. Used to associate the new `Operation` with an existing `Operation` (for example, a replica with the op that was replicated).
- `op_def`: Optional. The `op_def_pb2.OpDef` proto that describes the op type that this `Operation`represents.

Raises:

- `TypeError`: if control inputs are not Operations or Tensors, or if `node_def` is not a `NodeDef`, or if `g` is not a`Graph`, or if `inputs` are not tensors, or if `inputs` and `input_types` are incompatible.
- `ValueError`: if the `node_def` name is not valid.

`tf.Operation.node_def`

Returns a serialized `NodeDef` representation of this operation.

Returns:

A `NodeDef` protocol buffer.

`tf.Operation.op_def`

Returns the `OpDef` proto that represents the type of this op.

Returns:

An `OpDef` protocol buffer.

`tf.Operation.values()`

DEPRECATED: Use outputs.

`class tf.Tensor`

Represents a value produced by an `Operation`.

A `Tensor` is a symbolic handle to one of the outputs of an `Operation`. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow `Session`.

This class has two primary purposes:

1. A `Tensor` can be passed as an input to another `Operation`. This builds a dataflow connection between operations, which enables TensorFlow to execute an entire `Graph` that represents a large, multi-step computation.
2. After the graph has been launched in a session, the value of the `Tensor` can be computed by passing it

to`Session.run()`. `t.eval()` is a shortcut for

calling `tf.get_default_session().run(t)`.

In the following example, `c`, `d`, and `e` are symbolic `Tensor` objects,

whereas `result` is a numpy array that stores a concrete value:

```
# Build a dataflow graph.
c = tf.constant([[1.0, 2.0], [3.0, 4.0]])
d = tf.constant([[1.0, 1.0], [0.0, 1.0]])
e = tf.matmul(c, d)

# Construct a `Session` to execute the graph.
sess = tf.Session()

# Execute the graph and store the value that `e` represents in
`result`.
result = sess.run(e)
```

`tf.Tensor.dtype`

The `DType` of elements in this tensor.

`tf.Tensor.name`

The string name of this tensor.

`tf.Tensor.value_index`

The index of this tensor in the outputs of its `Operation`.

`tf.Tensor.graph`

The `Graph` that contains this tensor.

`tf.Tensor.op`

The `Operation` that produces this tensor as an output.

`tf.Tensor.consumers()`

Returns a list of `Operation`s that consume this tensor.

Returns:

A list of `Operation`s.

`tf.Tensor.eval(feed_dict=None, session=None)`

Evaluates this tensor in a `Session`.

Calling this method will execute all preceding operations that produce the inputs needed for the operation that produces this tensor.

N.B. Before invoking `Tensor.eval()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

Args:

- `feed_dict`: A dictionary that maps `Tensor` objects to feed values. See `Session.run()` for a description of the valid feed values.
- `session`: (Optional.) The `Session` to be used to evaluate this tensor. If none, the default session will be used.

Returns:

A numpy array corresponding to the value of this tensor.

`tf.Tensor.get_shape()`

Returns the `TensorShape` that represents the shape of this tensor.

The shape is computed using shape inference functions that are registered for each `Operation` type using`tf.RegisterShape`.

See `TensorShape` for more details of what a shape represents.

The inferred shape of a tensor is used to provide shape information without having to launch the graph in a session. This can be used for debugging, and providing early error messages. For example:

```
c = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

print(c.get_shape())
==> TensorShape([Dimension(2), Dimension(3)])

d = tf.constant([[1.0, 0.0], [0.0, 1.0], [1.0, 0.0], [0.0, 1.0]])

print(d.get_shape())
==> TensorShape([Dimension(4), Dimension(2)])

# Raises a ValueError, because `c` and `d` do not have compatible
# inner dimensions.
e = tf.matmul(c, d)
```

```
f = tf.matmul(c, d, transpose_a=True, transpose_b=True)

print(f.get_shape())
==> TensorShape([Dimension(3), Dimension(4)])
```

In some cases, the inferred shape may have unknown dimensions. If the caller has additional information about the values of these dimensions, `Tensor.set_shape()` can be used to augment the inferred shape.

Returns:

A `TensorShape` representing the shape of this tensor.

`tf.Tensor.set_shape(shape)`

Updates the shape of this tensor.

This method can be called multiple times, and will merge the given `shape` with the current shape of this tensor. It can be used to provide additional information about the shape of this tensor that cannot be inferred from the graph alone. For example, this can be used to provide additional information about the shapes of images:

```
_, image_data = tf.TFRecordReader(...).read(...)
image = tf.image.decode_png(image_data, channels=3)

# The height and width dimensions of `image` are data dependent,
and
# cannot be computed without executing the op.
print(image.get_shape())
==> TensorShape([Dimension(None), Dimension(None), Dimension(3)])

# We know that each image in this dataset is 28 x 28 pixels.
image.set_shape([28, 28, 3])
print(image.get_shape())
==> TensorShape([Dimension(28), Dimension(28), Dimension(3)])
```

Args:

- `shape`: A `TensorShape` representing the shape of this tensor.

Raises:

- `ValueError`: If `shape` is not compatible with the current shape of this tensor.

Other Methods

`tf.Tensor.__init__(op, value_index, dtype)`

Creates a new `Tensor`.

Args:

- `op`: An `Operation`. `Operation` that computes this tensor.
- `value_index`: An `int`. Index of the operation's endpoint that produces this tensor.
- `dtype`: A `DType`. Type of elements stored in this tensor.

Raises:

- `TypeError`: If the op is not an `Operation`.

`tf.Tensor.device`

The name of the device on which this tensor will be produced, or None.

Tensor types

`class tf.DType`

Represents the type of the elements in a `Tensor`.

The following `DType` objects are defined:

- `tf.float32`: 32-bit single-precision floating-point.
- `tf.float64`: 64-bit double-precision floating-point.
- `tf.bfloat16`: 16-bit truncated floating-point.
- `tf.complex64`: 64-bit single-precision complex.
- `tf.int8`: 8-bit signed integer.
- `tf.uint8`: 8-bit unsigned integer.
- `tf.uint16`: 16-bit unsigned integer.
- `tf.int16`: 16-bit signed integer.
- `tf.int32`: 32-bit signed integer.
- `tf.int64`: 64-bit signed integer.
- `tf.bool`: Boolean.
- `tf.string`: String.
- `tf.qint8`: Quantized 8-bit signed integer.
- `tf.quint8`: Quantized 8-bit unsigned integer.

- `tf.qint16`: Quantized 16-bit signed integer.
- `tf.quint16`: Quantized 16-bit unsigned integer.
- `tf.qint32`: Quantized 32-bit signed integer.

In addition, variants of these types with the `_ref` suffix are defined for reference-typed tensors.
The `tf.as_dtype()` function converts numpy types and string type names to a `DType` object.

`tf.DType.is_compatible_with(other)`

Returns True if the `other` DType will be converted to this DType.

The conversion rules are as follows:

```
DType(T)       .is_compatible_with(DType(T))        == True
DType(T)       .is_compatible_with(DType(T).as_ref)  == True
DType(T).as_ref.is_compatible_with(DType(T))        == False
DType(T).as_ref.is_compatible_with(DType(T).as_ref)  == True
```

Args:

- `other`: A `DType` (or object that may be converted to a `DType`).

Returns:

True if a Tensor of the `other` `DType` will be implicitly converted to this `DType`.

`tf.DType.name`

Returns the string name for this `DType`.

`tf.DType.base_dtype`

Returns a non-reference `DType` based on this `DType`.

`tf.DType.is_ref_dtype`

Returns `True` if this `DType` represents a reference type.

`tf.DType.as_ref`

Returns a reference `DType` based on this `DType`.

`tf.DType.is_floating`

Returns whether this is a (real) floating point type.

`tf.DType.is_integer`

Returns whether this is a (non-quantized) integer type.

`tf.DType.is_quantized`

Returns whether this is a quantized data type.

`tf.DType.is_unsigned`

Returns whether this type is unsigned.

Non-numeric, unordered, and quantized types are not considered unsigned, and this function returns `False`.

Returns:

Whether a `DType` is unsigned.

`tf.DType.as_numpy_dtype`

Returns a `numpy.dtype` based on this `DType`.

`tf.DType.as_datatype_enum`

Returns a `types_pb2.DataType` enum value based on this `DType`.

Other Methods

`tf.DType.__init__(type_enum)`

Creates a new `DataType`.

NOTE(mrry): In normal circumstances, you should not need to construct a `DataType` object directly. Instead, use the `tf.as_dtype()` function.

Args:

* `type_enum`: A `types_pb2.DataType` enum value.

Raises:

* `TypeError`: If `type_enum` is not a value `types_pb2.DataType`.

`tf.DType.max`

Returns the maximum representable value in this data type.

Raises:

* `TypeError`: if this is a non-numeric, unordered, or quantized type.

`tf.DType.min`

Returns the minimum representable value in this data type.

Raises:

- `TypeError`: if this is a non-numeric, unordered, or quantized type.

`tf.as_dtype(type_value)`

Converts the given `type_value` to a `DType`.

Args:

- `type_value`: A value that can be converted to a `tf.DType` object.

This may currently be a `tf.DType`object, a `DataType` enum, a string type name, or a `numpy.dtype`.

Returns:

A `DType` corresponding to `type_value`.

Raises:

- `TypeError`: If `type_value` cannot be converted to a `DType`.

Utility functions

`tf.device(dev)`

Wrapper for `Graph.device()` using the default graph.

See `Graph.device()` for more details.

Args:

- `device_name_or_function`: The device name or function to use in the context.

Returns:

A context manager that specifies the default device to use for newly created ops.

`tf.name_scope(name)`

Wrapper for `Graph.name_scope()` using the default graph.

See `Graph.name_scope()` for more details.

Args:

- `name`: A name for the scope.

Returns:

A context manager that installs `name` as a new name scope in the default graph.

`tf.control_dependencies(control_inputs)`

Wrapper for `Graph.control_dependencies()` using the default graph.

See `Graph.control_dependencies()` for more details.

Args:

- `control_inputs`: A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies.

Returns:

A context manager that specifies control dependencies for all operations constructed within the context.

`tf.convert_to_tensor(value, dtype=None, name=None, as_ref=False)`

Converts the given `value` to a `Tensor`.

This function converts Python objects of various types to `Tensor` objects. It accepts `Tensor` objects, numpy arrays, Python lists, and Python scalars. For example:

```
import numpy as np
array = np.random.rand(32, 100, 100)

def my_func(arg):
  arg = tf.convert_to_tensor(arg, dtype=tf.float32)
  return tf.matmul(arg, arg) + arg

# The following calls are equivalent.
value_1 = my_func(tf.constant([[1.0, 2.0], [3.0, 4.0]]))
value_2 = my_func([[1.0, 2.0], [3.0, 4.0]])
```

```
value_3 = my_func(np.array([[1.0, 2.0], [3.0, 4.0]],
dtype=np.float32))
```

This function can be useful when composing a new operation in Python (such as `my_func` in the example above). All standard Python op constructors apply this function to each of their Tensor-valued inputs, which allows those ops to accept numpy arrays, Python lists, and scalars in addition to `Tensor` objects.

Args:

- `value`: An object whose type has a registered `Tensor` conversion function.
- `dtype`: Optional element type for the returned tensor. If missing, the type is inferred from the type of `value`.
- `name`: Optional name to use if a new `Tensor` is created.
- `as_ref`: True if we want the result as a ref tensor.

Returns:

A `Tensor` based on `value`.

Raises:

- `TypeError`: If no conversion function is registered for `value`.
- `RuntimeError`: If a registered conversion function returns an invalid value.

`tf.convert_to_tensor_or_indexed_slices(value, dtype=None, name=None, as_ref=False)`

Converts the given object to a `Tensor` or an `IndexedSlices`.

If `value` is an `IndexedSlices` it is returned unmodified. Otherwise, it is converted to a `Tensor` using`convert_to_tensor()`.

Args:

- `value`: An `IndexedSlices` or an object that can be consumed by `convert_to_tensor()`.
- `dtype`: (Optional.) The required `DType` of the returned `Tensor` or `IndexedSlices`.
- `name`: (Optional.) A name to use if a new `Tensor` is created.
- `as_ref`: True if the caller wants the results as ref tensors.

Returns:

An `Tensor` or an `IndexedSlices` based on `value`.

Raises:

- `ValueError`: If `dtype` does not match the element type of `value`.

`tf.get_default_graph()`

Returns the default graph for the current thread.

The returned graph will be the innermost graph on which a `Graph.as_default()` context has been entered, or a global default graph if none has been explicitly created.
NOTE: The default graph is a property of the current thread. If you create a new thread, and wish to use the default graph in that thread, you must explicitly add a `with g.as_default():` in that thread's function.

Returns:

The default `Graph` being used in the current thread.

`tf.reset_default_graph()`

Clears the default graph stack and resets the global default graph.

NOTE: The default graph is a property of the current thread. This function applies only to the current thread. Calling this function while a `tf.Session` or `tf.InteractiveSession` is active will result in undefined behavior. Using any previously created `tf.Operation` or `tf.Tensor` objects after calling this function will result in undefined behavior.

`tf.import_graph_def(graph_def, input_map=None, return_elements=None, name=None, op_dict=None)`

Imports the TensorFlow graph in `graph_def` into the Python `Graph`. This function provides a way to import a serialized TensorFlow `GraphDef` protocol buffer, and extract individual objects in the `GraphDef` as `Tensor` and `Operation` objects.

See `Graph.as_graph_def()` for a way to create a `GraphDef` proto.

Args:

- `graph_def`: A `GraphDef` proto containing operations to be imported into the default graph.
- `input_map`: A dictionary mapping input names (as strings)

  in `graph_def` to `Tensor` objects. The values of the named input tensors in the imported graph will be re-mapped to the respective `Tensor` values.

- `return_elements`: A list of strings containing operation names

  in `graph_def` that will be returned as`Operation` objects; and/or

  tensor names in `graph_def` that will be returned as `Tensor` objects.

- `name`: (Optional.) A prefix that will be prepended to the names

  in `graph_def`. Defaults to `"import"`.

- `op_dict`: (Optional.) A dictionary mapping op type names

  to `OpDef` protos. Must contain an `OpDef` proto for each op type named

  in `graph_def`. If omitted, uses the `OpDef` protos registered in the global registry.

Returns:

A list of `Operation` and/or `Tensor` objects from the imported graph,

corresponding to the names in`return_elements`.

Raises:

- `TypeError`: If `graph_def` is not a `GraphDef` proto, `input_map` is not a dictionary mapping strings to`Tensor` objects, or `return_elements` is not a list of strings.
- `ValueError`: If `input_map`, or `return_elements` contains names that do not appear in `graph_def`, or`graph_def` is not well-formed (e.g. it refers to an unknown tensor).

`tf.load_op_library(library_filename)`

Loads a TensorFlow plugin, containing custom ops and kernels.

Pass "library_filename" to a platform-specific mechanism for dynamically loading a library. The rules for determining the exact location of the library are platform-specific and are not documented here. Expects the symbols "RegisterOps", "RegisterKernels", and "GetOpList", to be defined in the library.

Args:

- `library_filename`: Path to the plugin. Relative or absolute filesystem path to a dynamic library file.

Returns:

A python module containing the Python wrappers for Ops defined in the plugin.

Raises:

* `RuntimeError`: when unable to load the library or get the python wrappers.

Graph collections

`tf.add_to_collection(name, value)`

Wrapper for `Graph.add_to_collection()` using the default graph.

See `Graph.add_to_collection()` for more details.

Args:

* `name`: The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
* `value`: The value to add to the collection.

`tf.get_collection(key, scope=None)`

Wrapper for `Graph.get_collection()` using the default graph.

See `Graph.get_collection()` for more details.

Args:

* `key`: The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.

- `scope`: (Optional.) If supplied, the resulting list is filtered to include only items whose name begins with this string.

Returns:

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. The list contains the values in the order under which they were collected.

`class tf.GraphKeys`

Standard names to use for graph collections.

The standard library uses various well-known names to collect and retrieve values associated with a graph. For example, the `tf.Optimizer` subclasses default to optimizing the variables collected under`tf.GraphKeys.TRAINABLE_VARIABLES` if none is specified, but it is also possible to pass an explicit list of variables.

The following standard keys are defined:

- `VARIABLES`: the `Variable` objects that comprise a model, and must be saved and restored together. See`tf.all_variables()` for more details.
- `TRAINABLE_VARIABLES`: the subset of `Variable` objects that will be trained by an optimizer. See`tf.trainable_variables()` for more details.
- `SUMMARIES`: the summary `Tensor` objects that have been created in the graph. See`tf.merge_all_summaries()` for more details.

- `QUEUE_RUNNERS`: the `QueueRunner` objects that are used to produce input for a computation. See`tf.start_queue_runners()` for more details.
- `MOVING_AVERAGE_VARIABLES`: the subset of `Variable` objects that will also keep moving averages. See`tf.moving_average_variables()` for more details.
- `REGULARIZATION_LOSSES`: regularization losses collected during graph construction.
- `WEIGHTS`: weights inside neural network layers
- `BIASES`: biases inside neural network layers
- `ACTIVATIONS`: activations of neural network layers

Defining new operations

`class tf.RegisterGradient`

A decorator for registering the gradient function for an op type.

This decorator is only used when defining a new op type. For an op with `m` inputs and `n` outputs, the gradient function is a function that takes the original `Operation` and `n` `Tensor` objects (representing the gradients with respect to each output of the op), and returns `m` `Tensor` objects (representing the partial gradients with respect to each input of the op).

For example, assuming that operations of type `"Sub"` take two inputs `x` and `y`, and return a single output `x - y`, the following gradient function would be registered:

```
@tf.RegisterGradient("Sub")
def _sub_grad(unused_op, grad):
  return grad, tf.neg(grad)
```

The decorator argument `op_type` is the string type of an operation. This corresponds to the `OpDef.name` field for the proto that defines the operation.

`tf.RegisterGradient.__init__(op_type)`

Creates a new decorator with `op_type` as the Operation type.

Args:

- `op_type`: The string type of an operation. This corresponds to the `OpDef.name` field for the proto that defines the operation.

`tf.NoGradient(op_type)`

Specifies that ops of type `op_type` do not have a defined gradient. This function is only used when defining a new op type. It may be used for ops such as `tf.size()` that are not differentiable. For example:

```
tf.NoGradient("Size")
```

Args:

- `op_type`: The string type of an operation. This corresponds to the `OpDef.name` field for the proto that defines the operation.

Raises:

- `TypeError`: If `op_type` is not a string.

`class tf.RegisterShape`

A decorator for registering the shape function for an op type.

This decorator is only used when defining a new op type. A shape function is a function from an `Operation`object to a list

of `TensorShape` objects, with one `TensorShape` for each output of the operation.
For example, assuming that operations of type `"Sub"` take two

inputs `x` and `y`, and return a single output `x - y`, all with the same shape, the following shape function would be registered:

```
@tf.RegisterShape("Sub")
def _sub_shape(op):
  return
[op.inputs[0].get_shape().merge_with(op.inputs[1].get_shape())]
```

The decorator argument `op_type` is the string type of an operation.

This corresponds to the `OpDef.name` field for the proto that defines the operation.

`tf.RegisterShape.__init__(op_type)`

Saves the `op_type` as the `Operation` type.

`class tf.TensorShape`

Represents the shape of a `Tensor`.

A `TensorShape` represents a possibly-partial shape specification for a `Tensor`. It may be one of the following:

- Fully-known shape: has a known number of dimensions and a known size for each dimension.
- Partially-known shape: has a known number of dimensions, and an unknown size for one or more dimension.
- Unknown shape: has an unknown number of dimensions, and an unknown size in all dimensions.

If a tensor is produced by an operation of type `"Foo"`, its shape may be inferred if there is a registered shape function for `"Foo"`.

See `tf.RegisterShape()` for details of shape functions and how to register them. Alternatively, the shape may be set explicitly using `Tensor.set_shape()`.

`tf.TensorShape.merge_with(other)`

Returns a `TensorShape` combining the information in `self` and `other`.

The dimensions in `self` and `other` are merged elementwise, according to the rules defined for`Dimension.merge_with()`.

Args:

- `other`: Another `TensorShape`.

Returns:

A `TensorShape` containing the combined information

of `self` and `other`.

Raises:

- `ValueError`: If `self` and `other` are not compatible.

`tf.TensorShape.concatenate(other)`

Returns the concatenation of the dimension in `self` and `other`.

N.B. If either `self` or `other` is completely unknown, concatenation

will discard information about the other shape. In future, we might support concatenation that preserves this information for use with slicing.

Args:

- `other`: Another `TensorShape`.

Returns:

A `TensorShape` whose dimensions are the concatenation of the

dimensions in `self` and `other`.

`tf.TensorShape.ndims`

Returns the rank of this shape, or None if it is unspecified.

`tf.TensorShape.dims`

Returns a list of Dimensions, or None if the shape is unspecified.

`tf.TensorShape.as_list()`

Returns a list of integers or None for each dimension.

Returns:

A list of integers or None for each dimension.

`tf.TensorShape.as_proto()`

Returns this shape as a `TensorShapeProto`.

`tf.TensorShape.is_compatible_with(other)`

Returns True iff `self` is compatible with `other`.

Two possibly-partially-defined shapes are compatible if there exists a fully-defined shape that both shapes can represent. Thus, compatibility allows the shape inference code to reason about partially-defined shapes. For example:

- TensorShape(None) is compatible with all shapes.
- TensorShape([None, None]) is compatible with all two-dimensional shapes, such as TensorShape([32, 784]), and also TensorShape(None). It is not compatible with, for example, TensorShape([None]) or TensorShape([None, None, None]).
- TensorShape([32, None]) is compatible with all two-dimensional shapes with size 32 in the 0th dimension, and also TensorShape([None, None]) and TensorShape(None). It is not compatible with, for example, TensorShape([32]), TensorShape([32, None, 1]) or TensorShape([64, None]).
- TensorShape([32, 784]) is compatible with itself, and also TensorShape([32, None]), TensorShape([None, 784]), TensorShape([None, None]) and TensorShape(None). It is not compatible with, for example, TensorShape([32, 1, 784]) or TensorShape([None]).

The compatibility relation is reflexive and symmetric, but not transitive. For example, TensorShape([32, 784]) is compatible with TensorShape(None), and TensorShape(None) is compatible with TensorShape([4, 4]), but TensorShape([32, 784]) is not compatible with TensorShape([4, 4]).

Args:

- `other`: Another TensorShape.

Returns:

True iff `self` is compatible with `other`.

`tf.TensorShape.is_fully_defined()`

Returns True iff `self` is fully defined in every dimension.

`tf.TensorShape.with_rank(rank)`

Returns a shape based on `self` with the given rank.

This method promotes a completely unknown shape to one with a known rank.

Args:

- `rank`: An integer.

Returns:

A shape that is at least as specific as `self` with the given rank.

Raises:

- `ValueError`: If `self` does not represent a shape with the given `rank`.

`tf.TensorShape.with_rank_at_least(rank)`

Returns a shape based on `self` with at least the given rank.

Args:

- `rank`: An integer.

Returns:

A shape that is at least as specific as `self` with at least the given rank.

Raises:

- `ValueError`: If `self` does not represent a shape with at least the given `rank`.

`tf.TensorShape.with_rank_at_most(rank)`

Returns a shape based on `self` with at most the given rank.

Args:

- `rank`: An integer.

Returns:

A shape that is at least as specific as `self` with at most the given rank.

Raises:

- `ValueError`: If `self` does not represent a shape with at most the given `rank`.

`tf.TensorShape.assert_has_rank(rank)`

Raises an exception if `self` is not compatible with the given `rank`.

Args:

- `rank`: An integer.

Raises:

- `ValueError`: If `self` does not represent a shape with the given `rank`.

`tf.TensorShape.assert_same_rank(other)`

Raises an exception if `self` and `other` do not have compatible ranks.

Args:

- `other`: Another `TensorShape`.

Raises:

- `ValueError`: If `self` and `other` do not represent shapes with the same rank.

`tf.TensorShape.assert_is_compatible_with(other)`

Raises exception if `self` and `other` do not represent the same shape.

This method can be used to assert that there exists a shape that both `self` and `other` represent.

Args:

- `other`: Another TensorShape.

Raises:

- `ValueError`: If `self` and `other` do not represent the same shape.

`tf.TensorShape.assert_is_fully_defined()`

Raises an exception if `self` is not fully defined in every dimension.

Raises:

- `ValueError`: If `self` does not have a known value for every dimension.

Other Methods

`tf.TensorShape.__init__(dims)`

Creates a new TensorShape with the given dimensions.

Args:

* `dims`: A list of Dimensions, or None if the shape is unspecified.
* `DEPRECATED`: A single integer is treated as a singleton list.

`tf.TensorShape.num_elements()`

Returns the total number of elements, or none for incomplete shapes.

`class tf.Dimension`

Represents the value of one dimension in a TensorShape.

`tf.Dimension.__init__(value)`

Creates a new Dimension with the given value.

`tf.Dimension.assert_is_compatible_with(other)`

Raises an exception if `other` is not compatible with this Dimension.

Args:

* `other`: Another Dimension.

Raises:

- `ValueError`: If `self` and `other` are not compatible (see is_compatible_with).

`tf.Dimension.is_compatible_with(other)`

Returns true if `other` is compatible with this Dimension.

Two known Dimensions are compatible if they have the same value. An unknown Dimension is compatible with all other Dimensions.

Args:

- `other`: Another Dimension.

Returns:

True if this Dimension and `other` are compatible.

`tf.Dimension.merge_with(other)`

Returns a Dimension that combines the information in `self` and `other`.

Dimensions are combined as follows:

```
Dimension(n)   .merge_with(Dimension(n))    == Dimension(n)
Dimension(n)   .merge_with(Dimension(None)) == Dimension(n)
Dimension(None).merge_with(Dimension(n))    == Dimension(n)
Dimension(None).merge_with(Dimension(None)) == Dimension(None)
Dimension(n)   .merge_with(Dimension(m)) raises ValueError for
n != m
```

Args:

- `other`: Another Dimension.

Returns:

A Dimension containing the combined information of `self` and `other`.

Raises:

- `ValueError`: If `self` and `other` are not compatible (see is_compatible_with).

`tf.Dimension.value`

The value of this dimension, or None if it is unknown.

`tf.op_scope(values, name, default_name=None)`

Returns a context manager for use when defining a Python op.

This context manager validates that the given `values` are from the same graph, ensures that that graph is the default graph, and pushes a name scope.

For example, to define a new Python op called `my_op`:

```
def my_op(a, b, c, name=None):
  with tf.op_scope([a, b, c], name, "MyOp") as scope:
    a = tf.convert_to_tensor(a, name="a")
    b = tf.convert_to_tensor(b, name="b")
    c = tf.convert_to_tensor(c, name="c")
```

```
  # Define some computation that uses `a`, `b`, and `c`.
  return foo_op(..., name=scope)
```

Args:

- `values`: The list of `Tensor` arguments that are passed to the op function.
- `name`: The name argument that is passed to the op function.
- `default_name`: The default name to use if the `name` argument is `None`.

Returns:

A context manager for use in defining Python ops. Yields the name scope.

Raises:

- `ValueError`: if neither `name` nor `default_name` is provided.

`tf.get_seed(op_seed)`

Returns the local seeds an operation should use given an op-specific seed.

Given operation-specific seed, `op_seed`, this helper function returns two seeds derived from graph-level and op-level seeds. Many random operations internally use the two seeds to allow user to change the seed globally for a graph, or for only specific operations. For details on how the graph-level seed interacts with op seeds, see `set_random_seed`.

Args:

- `op_seed`: integer.

Returns:

A tuple of two integers that should be used for the local seed of this operation.

For libraries building on TensorFlow

`tf.register_tensor_conversion_function(base_type, conversion_func, priority=100)`

Registers a function for converting objects of `base_type` to `Tensor`.

The conversion function must have the following signature:

```
def conversion_func(value, dtype=None, name=None, as_ref=False):
  # ...
```

It must return a `Tensor` with the given `dtype` if specified. If the conversion function creates a new `Tensor`, it should use the given `name` if specified. All exceptions will be propagated to the caller. If `as_ref` is true, the function must return a `Tensor` reference, such as a `Variable`.

NOTE: The conversion functions will execute in order of priority, followed by order of registration. To ensure that a conversion function `F` runs before another conversion function `G`, ensure that `F` is registered with a smaller priority than `G`.

Args:

* `base_type`: The base type or tuple of base types for all objects

  that `conversion_func` accepts.
* `conversion_func`: A function that converts instances

  of `base_type` to `Tensor`.
* `priority`: Optional integer that indicates the priority for applying this

  conversion function. Conversion functions with smaller priority values run earlier than conversion functions with larger priority values. Defaults to 100.

Raises:

* `TypeError`: If the arguments do not have the appropriate type.

Other Functions and Classes

`class tf.bytes`

str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

Constants, Sequences, and Random Values

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

Contents

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], 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)`

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`, or `complex64`.
- `name`: A name for the operation (optional).

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], 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)`

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`, or `complex64`.
- `name`: A name for the operation (optional).

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, then the tensor is a scalar (0-D) if `value` is a scalar, or 1-D otherwise.

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` of type `int32`. 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`. First entry in sequence. Defaults to 0.
- `limit`: A 0-D (scalar) of type `int32`. Upper limit of sequence, exclusive.
- `delta`: A 0-D `Tensor` (scalar) of type `int32`. Optional. Default is 1. Number that increments `start`.
- `name`: A name for the operation (optional).

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.
c = tf.constant([[1, 2], [3, 4], [5, 6]])
sess = tf.Session()
```

```
norm = tf.random_normal(c, seed=1234)
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.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.

Variables

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

Contents

* `tf.variable_op_scope(values, name, default_name, initializer=None)`
* `tf.variable_scope(name_or_scope, reuse=None, initializer=None)`
* `tf.constant_initializer(value=0.0, dtype=tf.float32)`
* `tf.random_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)`
* `tf.truncated_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)`
* `tf.random_uniform_initializer(minval=0.0, maxval=1.0, seed=None, dtype=tf.float32)`
* `tf.uniform_unit_scaling_initializer(factor=1.0, seed=None, dtype=tf.float32)`
* `tf.zeros_initializer(shape, dtype=tf.float32)`
* Sparse Variable Updates
* `tf.scatter_update(ref, indices, updates, use_locking=None, name=None)`
* `tf.scatter_add(ref, indices, updates, use_locking=None, name=None)`
* `tf.scatter_sub(ref, indices, updates, use_locking=None, name=None)`
* `tf.sparse_mask(a, mask_indices, name=None)`
* `class tf.IndexedSlices`

Variables

`class tf.Variable`

See the Variables How To for a high level overview.

A variable maintains state in the graph across calls to `run()`. You add a variable to the graph by constructing an instance of the class `Variable`.

The `Variable()` constructor requires an initial value for the variable, which can be a `Tensor` of any type and shape. The initial value defines the type and shape of the variable. After construction, the

type and shape of the variable are fixed. The value can be changed using one of the assign methods.
If you want to change the shape of a variable later you have to use an `assign` Op with`validate_shape=False`.

Just like any `Tensor`, variables created with `Variable()` can be used as inputs for other Ops in the graph. Additionally, all the operators overloaded for the `Tensor` class are carried over to variables, so you can also add nodes to the graph by just doing arithmetic on variables.

```
import tensorflow as tf

# Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

# Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

# The overloaded operators are available too.
z = tf.sigmoid(w + b)

# Assign a new value to the variable with `assign()` or a related
method.
w.assign(w + 1.0)
w.assign_add(1.0)
```

When you launch the graph, variables have to be explicitly initialized before you can run Ops that use their value. You can initialize a variable by running its initializer op, restoring the variable from a save file, or simply running an`assign` Op that assigns a value to the variable. In fact, the variable initializer op is just an `assign` Op that assigns the variable's initial value to the variable itself.

```
# Launch the graph in a session.
with tf.Session() as sess:
    # Run the variable initializer.
    sess.run(w.initializer)
    # ...you now can run ops that use the value of 'w'...
```

The most common initialization pattern is to use the convenience function `initialize_all_variables()` to add an Op to the graph that initializes all the variables. You then run that Op after launching the graph.

```
# Add an Op to initialize all variables.
init_op = tf.initialize_all_variables()

# Launch the graph in a session.
with tf.Session() as sess:
    # Run the Op that initializes all variables.
    sess.run(init_op)
    # ...you can now run any Op that uses variable values...
```

If you need to create a variable with an initial value dependent on another variable, use the other variable's`initialized_value()`. This ensures that variables are initialized in the right order.

All variables are automatically collected in the graph where they are created. By default, the constructor adds the new variable to the graph collection `GraphKeys.VARIABLES`. The convenience function `all_variables()`returns the contents of that collection.

When building a machine learning model it is often convenient to distinguish betwen variables holding the trainable model parameters and other variables such as a `global step` variable used to count training steps. To make this easier, the variable constructor supports a `trainable=<bool>` parameter. If `True`, the new variable is also added to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. The convenience function`trainable_variables()` returns the contents of this collection. The various `Optimizer` classes use this collection as the default list of variables to optimize.

Creating a variable.

```
tf.Variable.__init__(initial_value=None, trainable=True, collections=None, validate_shape=True, name=None, variable_def=None)
```

Creates a new variable with value `initial_value`.

The new variable is added to the graph collections listed in `collections`, which defaults to `[GraphKeys.VARIABLES]`.

If `trainable` is `True` the variable is also added to the graph collection`GraphKeys.TRAINABLE_VARIABLES`.

This constructor creates both a `variable` Op and an `assign` Op to set the variable to its initial value.

Args:

- `initial_value`: A `Tensor`, or Python object convertible to a `Tensor`. The initial value for the Variable. Must have a shape specified unless `validate_shape` is set to False.
- `trainable`: If `True`, the default, also adds the variable to the graph collection`GraphKeys.TRAINABLE_VARIABLES`. This collection is used as the default list of variables to use by the`Optimizer` classes.
- `collections`: List of graph collections keys. The new variable is added to these collections. Defaults to `[GraphKeys.VARIABLES]`.
- `validate_shape`: If `False`, allows the variable to be initialized with a value of unknown shape. If `True`, the default, the shape of `initial_value` must be known.
- `name`: Optional name for the variable. Defaults to `'Variable'` and gets uniquified automatically.
- `variable_def`: `VariableDef` protocol buffer. If not `None`, recreates the Variable object with its contents.`variable_def` and the other arguments are mutually exclusive.

Returns:

A Variable.

Raises:

- `ValueError`: If both `variable_def` and initial_value are specified.
- `ValueError`: If the initial value is not specified, or does not have a shape and `validate_shape` is `True`.

`tf.Variable.initialized_value()`

Returns the value of the initialized variable.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
# Initialize 'v' with a random tensor.
v = tf.Variable(tf.truncated_normal([10, 40]))
# Use `initialized_value` to guarantee that `v` has been
# initialized before its value is used to initialize `w`.
# The random values are picked only once.
w = tf.Variable(v.initialized_value() * 2.0)
```

Returns:

A `Tensor` holding the value of this variable after its initializer has run.

Changing a variable value.

`tf.Variable.assign(value, use_locking=False)`

Assigns a new value to the variable.

This is essentially a shortcut for `assign(self, value)`.

Args:

- `value`: A `Tensor`. The new value for this variable.
- `use_locking`: If `True`, use locking during the assignment.

Returns:

A `Tensor` that will hold the new value of this variable after the assignment has completed.

`tf.Variable.assign_add(delta, use_locking=False)`

Adds a value to this variable.

This is essentially a shortcut for `assign_add(self, delta)`.

Args:

- `delta`: A `Tensor`. The value to add to this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the addition has completed.

`tf.Variable.assign_sub(delta, use_locking=False)`

Subtracts a value from this variable.

This is essentially a shortcut for `assign_sub(self, delta)`.

Args:

- `delta`: A `Tensor`. The value to subtract from this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the subtraction has completed.

`tf.Variable.scatter_sub(sparse_delta, use_locking=False)`

Subtracts `IndexedSlices` from this variable.

This is essentially a shortcut for `scatter_sub(self, sparse_delta.indices, sparse_delta.values)`.

Args:

- `sparse_delta`: `IndexedSlices` to be subtracted from this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the scattered subtraction has completed.

Raises:

- `ValueError`: if `sparse_delta` is not an `IndexedSlices`.

`tf.Variable.count_up_to(limit)`

Increments this variable until it reaches `limit`.

When that Op is run it tries to increment the variable by `1`. If incrementing the variable would bring it above `limit`then the Op raises the exception `OutOfRangeError`.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for `count_up_to(self, limit)`.

Args:

- `limit`: value at which incrementing the variable raises an error.

Returns:

A `Tensor` that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

`tf.Variable.eval(session=None)`

In a session, computes and returns the value of this variable.

This is not a graph construction method, it does not add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See the Session class for more information on launching a graph and on sessions.

```
v = tf.Variable([1, 2])
init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    # Usage passing the session explicitly.
    print(v.eval(sess))
    # Usage with the default session.  The 'with' block
    # above makes 'sess' the default session.
    print(v.eval())
```

Args:

- `session`: The session to use to evaluate this variable. If none, the default session is used.

Returns:

A numpy `ndarray` with a copy of the value of this variable.

Properties.

`tf.Variable.name`

The name of this variable.

`tf.Variable.dtype`

The `DType` of this variable.

`tf.Variable.get_shape()`

The `TensorShape` of this variable.

Returns:

A `TensorShape`.

`tf.Variable.device`

The device of this variable.

`tf.Variable.initializer`

The initializer operation for this variable.

`tf.Variable.graph`

The `Graph` of this variable.

`tf.Variable.op`

The `Operation` of this variable.

Other Methods

`tf.Variable.from_proto(variable_def)`

`tf.Variable.ref()`

Returns a reference to this variable.

You usually do not need to call this method as all ops that need a reference to the variable call it automatically.

Returns is a `Tensor` which holds a reference to the variable. You can assign a new value to the variable by passing the tensor to an assign op. See `value()` if you want to get the value of the variable.

Returns:

A `Tensor` that is a reference to the variable.

`tf.Variable.to_proto()`

Converts a `Variable` to a `VariableDef` protocol buffer.

Returns:

A `VariableDef` protocol buffer.

`tf.Variable.value()`

Returns the last snapshot of this variable.

You usually do not need to call this method as all ops that need the value of the variable call it automatically through a `convert_to_tensor()` call.

Returns a `Tensor` which holds the value of the variable. You can not assign a new value to this tensor as it is not a reference to the variable. See `ref()` if you want to get a reference to the variable.

To avoid copies, if the consumer of the returned value is on the same device as the variable, this actually returns the live value of the variable, not a copy. Updates to the variable are seen by the consumer. If the consumer is on a different device it will get a copy of the variable.

Returns:

A `Tensor` containing the value of the variable.

Variable helper functions

TensorFlow provides a set of functions to help manage the set of variables collected in the graph.

`tf.all_variables()`

Returns all variables collected in the graph.

The `Variable()` constructor automatically adds new variables to the graph collection`GraphKeys.VARIABLES`. This convenience function returns the contents of that collection.

Returns:

A list of `Variable` objects.

`tf.trainable_variables()`

Returns all variables created with `trainable=True`.

When passed `trainable=True`, the `Variable()` constructor automatically adds new variables to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. This convenience function returns the contents of that collection.

Returns:

A list of Variable objects.

`tf.moving_average_variables()`

Returns all variables that maintain their moving averages.

If an `ExponentialMovingAverage` object is created and

the `apply()` method is called on a list of variables, these variables will be added to

the `GraphKeys.MOVING_AVERAGE_VARIABLES` collection. This convenience function returns the contents of that collection.

Returns:

A list of Variable objects.

`tf.initialize_all_variables()`

Returns an Op that initializes all variables.

This is just a shortcut for `initialize_variables(all_variables())`

Returns:

An Op that initializes all variables in the graph.

`tf.initialize_variables(var_list, name='init')`

Returns an Op that initializes a list of variables.

After you launch the graph in a session, you can run the returned Op to initialize all the variables in `var_list`. This Op runs all the initializers of the variables in `var_list` in parallel.

Calling `initialize_variables()` is equivalent to passing the list of initializers to `Group()`.

If `var_list` is empty, however, the function still returns an Op that can be run. That Op just has no effect.

Args:

- `var_list`: List of `Variable` objects to initialize.
- `name`: Optional name for the returned operation.

Returns:

An Op that run the initializers of all the specified variables.

`tf.assert_variables_initialized(var_list=None)`

Returns an Op to check if variables are initialized.

When run, the returned Op will raise the exception `FailedPreconditionError` if any of the variables has not yet been initialized.

Note: This function is implemented by trying to fetch the values of the variables. If one of the variables is not initialized a message may be logged by the C++ runtime. This is expected.

Args:

- `var_list`: List of `Variable` objects to check. Defaults to the value of `all_variables()`.

Returns:

An Op, or None if there are no variables.

Saving and Restoring Variables

`class tf.train.Saver`

Saves and restores variables.

See [Variables] for an overview of variables, saving and restoring.

The `Saver` class adds ops to save and restore variables to and from checkpoints. It also provides convenience methods to run these ops.
Checkpoints are binary files in a proprietary format which map variable names to tensor values. The best way to examine the contents of a checkpoint is to load it using a `Saver`.

Savers can automatically number checkpoint filenames with a provided counter. This lets you keep multiple checkpoints at different steps while training a model. For example you can number the checkpoint filenames with the training step number. To avoid filling up disks, savers manage checkpoint files automatically. For example, they can keep only the N most recent files, or one checkpoint for every N hours of training.

You number checkpoint filenames by passing a value to the optional `global_step` argument to `save()`:

```
saver.save(sess, 'my-model', global_step=0) ==> filename: 'my-model-0'
```

```
...
saver.save(sess, 'my-model', global_step=1000) ==> filename: 'my-model-1000'
```

Additionally, optional arguments to the `Saver()` constructor let you control the proliferation of checkpoint files on disk:

- `max_to_keep` indicates the maximum number of recent checkpoint files to keep. As new files are created, older files are deleted. If None or 0, all checkpoint files are kept. Defaults to 5 (that is, the 5 most recent checkpoint files are kept.)
- `keep_checkpoint_every_n_hours`: In addition to keeping the most recent `max_to_keep` checkpoint files, you might want to keep one checkpoint file for every N hours of training. This can be useful if you want to later analyze how a model progressed during a long training session. For example, passing`keep_checkpoint_every_n_hours=2` ensures that you keep one checkpoint file for every 2 hours of training. The default value of 10,000 hours effectively disables the feature.

Note that you still have to call the `save()` method to save the model. Passing these arguments to the constructor will not save variables automatically for you.

A training program that saves regularly looks like:

```
...
# Create a saver.
saver = tf.train.Saver(...variables...)
# Launch the graph and train, saving the model every 1,000 steps.
sess = tf.Session()
for step in xrange(1000000):
    sess.run(..training_op..)
    if step % 1000 == 0:
        # Append the step number to the checkpoint name:
        saver.save(sess, 'my-model', global_step=step)
```

In addition to checkpoint files, savers keep a protocol buffer on disk with the list of recent checkpoints. This is used to manage numbered checkpoint files and by `latest_checkpoint()`, which makes it easy to discover the path to the most recent checkpoint. That protocol

buffer is stored in a file named 'checkpoint' next to the checkpoint files.
If you create several savers, you can specify a different filename for the protocol buffer file in the call to `save()`.

```
tf.train.Saver.__init__(var_list=None, reshape=False, sharded=False, max_to_keep=5, keep_checkpoint_every_n_hours=10000.0, name=None, restore_sequentially=False, saver_def=None, builder=None)
```

Creates a `Saver`.

The constructor adds ops to save and restore variables.

`var_list` specifies the variables that will be saved and restored. It can be passed as a `dict` or a list:

- A `dict` of names to variables: The keys are the names that will be used to save or restore the variables in the checkpoint files.
- A list of variables: The variables will be keyed with their op name in the checkpoint files.

For example:

```
v1 = tf.Variable(..., name='v1')
v2 = tf.Variable(..., name='v2')

# Pass the variables as a dict:
saver = tf.train.Saver({'v1': v1, 'v2': v2})

# Or pass them as a list.
saver = tf.train.Saver([v1, v2])
# Passing a list is equivalent to passing a dict with the
variable op names
# as keys:
saver = tf.train.Saver({v.op.name: v for v in [v1, v2]})
```

The optional `reshape` argument, if `True`, allows restoring a variable from a save file where the variable had a different shape, but the same number of elements and type. This is useful if you have reshaped a variable and want to reload it from an older checkpoint.

The optional `sharded` argument, if `True`, instructs the saver to shard checkpoints per device.

Args:

- `var_list`: A list of `Variable` objects or a dictionary mapping names to variables. If `None`, defaults to the list of all variables.
- `reshape`: If `True`, allows restoring parameters from a checkpoint where the variables have a different shape.
- `sharded`: If `True`, shard the checkpoints, one per device.
- `max_to_keep`: Maximum number of recent checkpoints to keep. Defaults to 5.
- `keep_checkpoint_every_n_hours`: How often to keep checkpoints. Defaults to 10,000 hours.
- `name`: String. Optional name to use as a prefix when adding operations.
- `restore_sequentially`: A `Bool`, which if true, causes restore of different variables to happen sequentially within each device. This can lower memory usage when restoring very large models.
- `saver_def`: Optional `SaverDef` proto to use instead of running the builder. This is only useful for specialty code that wants to recreate a `Saver` object for a previously built `Graph` that had a `Saver`.

  The `saver_def`proto should be the one returned by

  the `as_saver_def()` call of the `Saver` that was created for that`Graph`.

- `builder`: Optional `SaverBuilder` to use if a `saver_def` was not provided. Defaults to`BaseSaverBuilder()`.

Raises:

- `TypeError`: If `var_list` is invalid.
- `ValueError`: If any of the keys or values in `var_list` are not unique.

```
tf.train.Saver.save(sess, save_path, global_step=None,
latest_filename=None, meta_graph_suffix='meta')
```

Saves variables.

This method runs the ops added by the constructor for saving variables. It requires a session in which the graph was launched. The variables to save must also have been initialized.

The method returns the path of the newly created checkpoint file. This path can be passed directly to a call to`restore()`.

Args:

- `sess`: A Session to use to save the variables.
- `save_path`: String. Path to the checkpoint filename. If the saver is `sharded`, this is the prefix of the sharded checkpoint filename.
- `global_step`: If provided the global step number is appended to `save_path` to create the checkpoint filename. The optional argument can be a `Tensor`, a `Tensor` name or an integer.

- `latest_filename`: Optional name for the protocol buffer file that will contains the list of most recent checkpoint filenames. That file, kept in the same directory as the checkpoint files, is automatically managed by the saver to keep track of recent checkpoints. Defaults to 'checkpoint'.
- `meta_graph_suffix`: Suffix for MetaGraphDef file. Defaults to 'meta'.

Returns:

A string: path at which the variables were saved. If the saver is sharded, this string ends with: '-?????-of-nnnnn' where 'nnnnn' is the number of shards created.

Raises:

- `TypeError`: If `sess` is not a `Session`.
- `ValueError`: If `latest_filename` contains path components.

`tf.train.Saver.restore(sess, save_path)`

Restores previously saved variables.

This method runs the ops added by the constructor for restoring variables. It requires a session in which the graph was launched. The variables to restore do not have to have been initialized, as restoring is itself a way to initialize variables.

The `save_path` argument is typically a value previously returned from a `save()` call, or a call to`latest_checkpoint()`.

Args:

* `sess`: A `Session` to use to restore the parameters.
* `save_path`: Path where parameters were previously saved.

Other utility methods.

`tf.train.Saver.last_checkpoints`

List of not-yet-deleted checkpoint filenames.

You can pass any of the returned values to `restore()`.

Returns:

A list of checkpoint filenames, sorted from oldest to newest.

`tf.train.Saver.set_last_checkpoints(last_checkpoints)`

DEPRECATED: Use set_last_checkpoints_with_time.

Sets the list of old checkpoint filenames.

Args:

* `last_checkpoints`: A list of checkpoint filenames.

Raises:

* `AssertionError`: If last_checkpoints is not a list.

`tf.train.Saver.as_saver_def()`

Generates a `SaverDef` representation of this saver.

Returns:

A `SaverDef` proto.

Other Methods

`tf.train.Saver.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.

`tf.train.Saver.from_proto(saver_def)`

`tf.train.Saver.set_last_checkpoints_with_time(last_checkpoints_with_time)`

Sets the list of old checkpoint filenames and timestamps.

Args:

- `last_checkpoints_with_time`: A list of tuples of checkpoint filenames and timestamps.

Raises:

- `AssertionError`: If last_checkpoints_with_time is not a list.

`tf.train.Saver.to_proto()`

Returns a `SaverDef` protocol buffer.

`tf.train.latest_checkpoint(checkpoint_dir, latest_filename=None)`

Finds the filename of latest saved checkpoint file.

Args:

- `checkpoint_dir`: Directory where the variables were saved.

- `latest_filename`: Optional name for the protocol buffer file that contains the list of most recent checkpoint filenames. See the corresponding argument to `Saver.save()`.

Returns:

The full path to the latest checkpoint or `None` if no checkpoint was found.

`tf.train.get_checkpoint_state(checkpoint_dir, latest_filename=None)`

Returns CheckpointState proto from the "checkpoint" file.

If the "checkpoint" file contains a valid CheckpointState proto, returns it.

Args:

- `checkpoint_dir`: The directory of checkpoints.
- `latest_filename`: Optional name of the checkpoint file. Default to 'checkpoint'.

Returns:

A CheckpointState if the state was available, None otherwise.

```
tf.train.update_checkpoint_state(save_dir,
model_checkpoint_path, all_model_checkpoint_paths=None,
latest_filename=None)
```

Updates the content of the 'checkpoint' file.

This updates the checkpoint file containing a CheckpointState proto.

Args:

- `save_dir`: Directory where the model was saved.
- `model_checkpoint_path`: The checkpoint file.
- `all_model_checkpoint_paths`: List of strings. Paths to all not-yet-deleted checkpoints, sorted from oldest to newest. If this is a non-empty list, the last element must be equal to model_checkpoint_path. These paths are also saved in the CheckpointState proto.
- `latest_filename`: Optional name of the checkpoint file. Default to 'checkpoint'.

Raises:

- `RuntimeError`: If the save paths conflict.

Sharing Variables

TensorFlow provides several classes and operations that you can use to create variables contingent on certain conditions.

```
tf.get_variable(name, shape=None, dtype=tf.float32,
initializer=None, trainable=True, collections=None)
```

Gets an existing variable with these parameters or create a new one.

This function prefixes the name with the current variable scope and performs reuse checks. See the Variable Scope How To for an extensive description of how reusing works. Here is a basic example:

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])  # v.name == "foo/v:0"
    w = tf.get_variable("w", [1])  # w.name == "foo/w:0"
with tf.variable_scope("foo", reuse=True)
    v1 = tf.get_variable("v")  # The same as v above.
```

If initializer is `None` (the default), the default initializer passed in the constructor is used. If that one is `None` too, a`UniformUnitScalingInitializer` will be used. The initializer can also be a Tensor, in which case the variable is initialized to this value and shape.

Args:

- `name`: the name of the new or existing variable.
- `shape`: shape of the new or existing variable.
- `dtype`: type of the new or existing variable (defaults to `DT_FLOAT`).
- `initializer`: initializer for the variable if one is created.
- `trainable`: If `True` also add the variable to the graph collection `GraphKeys.TRAINABLE_VARIABLES`(see tf.Variable).
- `collections`: List of graph collections keys to add the Variable to. Defaults to `[GraphKeys.VARIABLES]`(see tf.Variable).

Returns:

The created or existing variable.

Raises:

* `ValueError`: when creating a new variable and shape is not declared, or when violating reuse during variable creation. Reuse is set inside `variable_scope`.

`tf.get_variable_scope()`

Returns the current variable scope.

`tf.make_template(name_, func_, **kwargs)`

Given an arbitrary function, wrap it so that it does variable sharing.

This wraps `func_` in a Template and partially evaluates it. Templates are functions that create variables the first time they are called and reuse them thereafter. In order for `func_` to be compatible with

a `Template` it must have the following properties:

* The function should create all trainable variables and any variables that should be reused by calling`tf.get_variable`. If a trainable

  variable is created using `tf.Variable`, then a ValueError will be thrown. Variables that are intended to be locals can be created by specifying `tf.Variable(..., trainable=false).`
* The function may use variable scopes and other templates internally to create and reuse variables, but it shouldn't

  use `tf.get_variables` to capture variables that are defined outside of the scope of the function.
* Internal scopes and variable names should not depend on any arguments that are not supplied to`make_template`. In general you will

get a ValueError telling you that you are trying to reuse a variable that doesn't exist if you make a mistake.

In the following example, both `z` and `w` will be scaled by the same `y`. It is important to note that if we didn't assign`scalar_name` and used a different name for z and w that a `ValueError` would be thrown because it couldn't reuse the variable.

```
def my_op(x, scalar_name):
  var1 = tf.get_variable(scalar_name,
                         shape=[],
                         initializer=tf.constant_initializer(1))
  return x * var1

scale_by_y = tf.make_template('scale_by_y', my_op,
scalar_name='y')

z = scale_by_y(input1)
w = scale_by_y(input2)
```

As a safe-guard, the returned function will raise a `ValueError` after the first call if trainable variables are created by calling `tf.Variable`.

If all of these are true, then 2 properties are enforced by the template:

1. Calling the same template multiple times will share all non-local variables.
2. Two different templates are guaranteed to be unique, unless you reenter the same variable scope as the initial definition of a template and redefine it. An examples of this exception:

```
def my_op(x, scalar_name):
  var1 = tf.get_variable(scalar_name,
                         shape=[],
                         initializer=tf.constant_initializer(1))
  return x * var1

with tf.variable_scope('scope') as vs:
  scale_by_y = tf.make_template('scale_by_y', my_op,
scalar_name='y')
  z = scale_by_y(input1)
  w = scale_by_y(input2)
```

```
# Creates a template that reuses the variables above.
with tf.variable_scope(vs, reuse=True):
  scale_by_y2 = tf.make_template('scale_by_y', my_op,
scalar_name='y')
  z2 = scale_by_y2(input1)
  w2 = scale_by_y2(input2)
```

Note: The full variable scope is captured at the time of the first call.

Note: `name_` and `func_` have a following underscore to reduce the likelihood of collisions with kwargs.

Args:

- `name_`: A name for the scope created by this template. If necessary, the name will be made unique by appending `_N` to the name.
- `func_`: The function to wrap.
- `**kwargs`: Keyword arguments to apply to `func_`.

Returns:

A function that will enter a `variable_scope` before calling `func_`. The first time it is called, it will create a non-reusing scope so that the variables will be unique. On each subsequent call, it will reuse those variables.

Raises:

- `ValueError`: if the name is None.

`tf.variable_op_scope(values, name, default_name, initializer=None)`

Returns a context manager for defining an op that creates variables.

This context manager validates that the given `values` are from the same graph, ensures that that graph is the default graph, and pushes a name scope and a variable scope.

If `name` is not None, it is used as is in the variable scope. If `name` is None, then `default_name` is used. In that case, if the same name has been previously used in the same scope, it will made unique be appending `_N` to it.

This is intended to be used when defining generic ops and so reuse is always inherited.

For example, to define a new Python op called `my_op_with_vars`:

```
def my_op_with_vars(a, b, name=None):
  with tf.variable_op_scope([a, b], name, "MyOp") as scope:
    a = tf.convert_to_tensor(a, name="a")
    b = tf.convert_to_tensor(b, name="b")
    c = tf.get_variable('c')
    # Define some computation that uses `a`, `b`, and `c`.
    return foo_op(..., name=scope)
```

Args:

- `values`: The list of `Tensor` arguments that are passed to the op function.
- `name`: The name argument that is passed to the op function, this name is not uniquified in the variable scope.
- `default_name`: The default name to use if the `name` argument is `None`, this name will be uniquified.
- `initializer`: A default initializer to pass to variable scope.

Returns:

A context manager for use in defining a Python op.

Raises:

- `ValueError`: when trying to reuse within a create scope, or create within a reuse scope, or if reuse is not`None` or `True`.
- `TypeError`: when the types of some arguments are not appropriate.

`tf.variable_scope(name_or_scope, reuse=None, initializer=None)`

Returns a context for variable scope.

Variable scope allows to create new variables and to share already created ones while providing checks to not create or share by accident. For details, see the Variable Scope How To, here we present only a few basic examples.

Simple example of how to create a new variable:

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
        assert v.name == "foo/bar/v:0"
```

Basic example of sharing a variable:

```
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 == v
```

Sharing a variable by capturing a scope and setting reuse:

```
with tf.variable_scope("foo") as scope:
    v = tf.get_variable("v", [1])
    scope.reuse_variables()
    v1 = tf.get_variable("v", [1])
assert v1 == v
```

To prevent accidental sharing of variables, we raise an exception when getting an existing variable in a non-reusing scope.

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
    v1 = tf.get_variable("v", [1])
    #  Raises ValueError("... v already exists ...").
```

Similarly, we raise an exception when trying to get a variable that does not exist in reuse mode.

```
with tf.variable_scope("foo", reuse=True):
    v = tf.get_variable("v", [1])
    #  Raises ValueError("... v does not exists ...").
```

Note that the `reuse` flag is inherited: if we open a reusing scope, then all its sub-scopes become reusing as well.

Args:

- `name_or_scope`: `string` or `VariableScope`: the scope to open.
- `reuse`: `True` or `None`; if `True`, we go into reuse mode for this scope as well as all sub-scopes; if `None`, we just inherit the parent scope reuse.
- `initializer`: default initializer for variables within this scope.

Returns:

A scope that can be to captured and reused.

Raises:

* `ValueError`: when trying to reuse within a create scope, or create within a reuse scope, or if reuse is not`None` or `True`.
* `TypeError`: when the types of some arguments are not appropriate.

`tf.constant_initializer(value=0.0, dtype=tf.float32)`

Returns an initializer that generates tensors with a single value.

Args:

* `value`: A Python scalar. All elements of the initialized variable will be set to this value.
* `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with a single value.

Raises:

* `ValueError`: if `dtype` is not a floating point type.

`tf.random_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)`

Returns an initializer that generates tensors with a normal distribution.

Args:

- `mean`: a python scalar or a scalar tensor. Mean of the random values to generate.
- `stddev`: a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with a normal distribution.

Raises:

- `ValueError`: if `dtype` is not a floating point type.

```
tf.truncated_normal_initializer(mean=0.0, stddev=1.0,
seed=None, dtype=tf.float32)
```

Returns an initializer that generates a truncated normal distribution.

These values are similar to values from a `random_normal_initializer` except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

Args:

- `mean`: a python scalar or a scalar tensor. Mean of the random values to generate.
- `stddev`: a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with a truncated normal distribution.

Raises:

- `ValueError`: if `dtype` is not a floating point type.

```
tf.random_uniform_initializer(minval=0.0, maxval=1.0,
seed=None, dtype=tf.float32)
```

Returns an initializer that generates tensors with a uniform distribution.

Args:

- `minval`: a python scalar or a scalar tensor. lower bound of the range of random values to generate.
- `maxval`: a python scalar or a scalar tensor. upper bound of the range of random values to generate.

- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.

- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with a uniform distribution.

Raises:

- `ValueError`: if `dtype` is not a floating point type.

`tf.uniform_unit_scaling_initializer(factor=1.0, seed=None, dtype=tf.float32)`

Returns an initializer that generates tensors without scaling variance.

When initializing a deep network, it is in principle advantageous to keep the scale of the input variance constant, so it does not explode or diminish by reaching the final layer. If the input is `x` and the operation `x * W`, and we want to initialize `W` uniformly at random, we need to pick `W` from

```
[-sqrt(3) / sqrt(dim), sqrt(3) / sqrt(dim)]
```

to keep the scale intact, where `dim = W.shape[0]` (the size of the input). A similar calculation for convolutional networks gives an analogous result with `dim` equal to the product of the first 3 dimensions. When nonlinearities are present, we need to multiply this by a constant `factor`. See Sussillo et al., 2014 (pdf) for deeper motivation, experiments and the calculation of constants. In section

2.3 there, the constants were numerically computed: for a linear layer it's 1.0, relu: ~1.43, tanh: ~1.15.

Args:

- `factor`: Float. A multiplicative factor by which the values will be scaled.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with unit variance.

Raises:

- `ValueError`: if `dtype` is not a floating point type.

`tf.zeros_initializer(shape, dtype=tf.float32)`

An adaptor for zeros() to match the Initializer spec.

Sparse Variable Updates

The sparse update ops modify a subset of the entries in a dense `Variable`, either overwriting the entries or adding / subtracting a delta. These are useful for training embedding models and similar lookup-based networks, since only a small subset of embedding vectors change in any given step.

Since a sparse update of a large tensor may be generated automatically during gradient computation (as in the gradient of `tf.gather`), an `IndexedSlices` class is provided that encapsulates a set of sparse indices and values. `IndexedSlices` objects are detected and handled automatically by the optimizers in most cases.

`tf.scatter_update(ref, indices, updates, use_locking=None, name=None)`

Applies sparse updates to a variable reference.

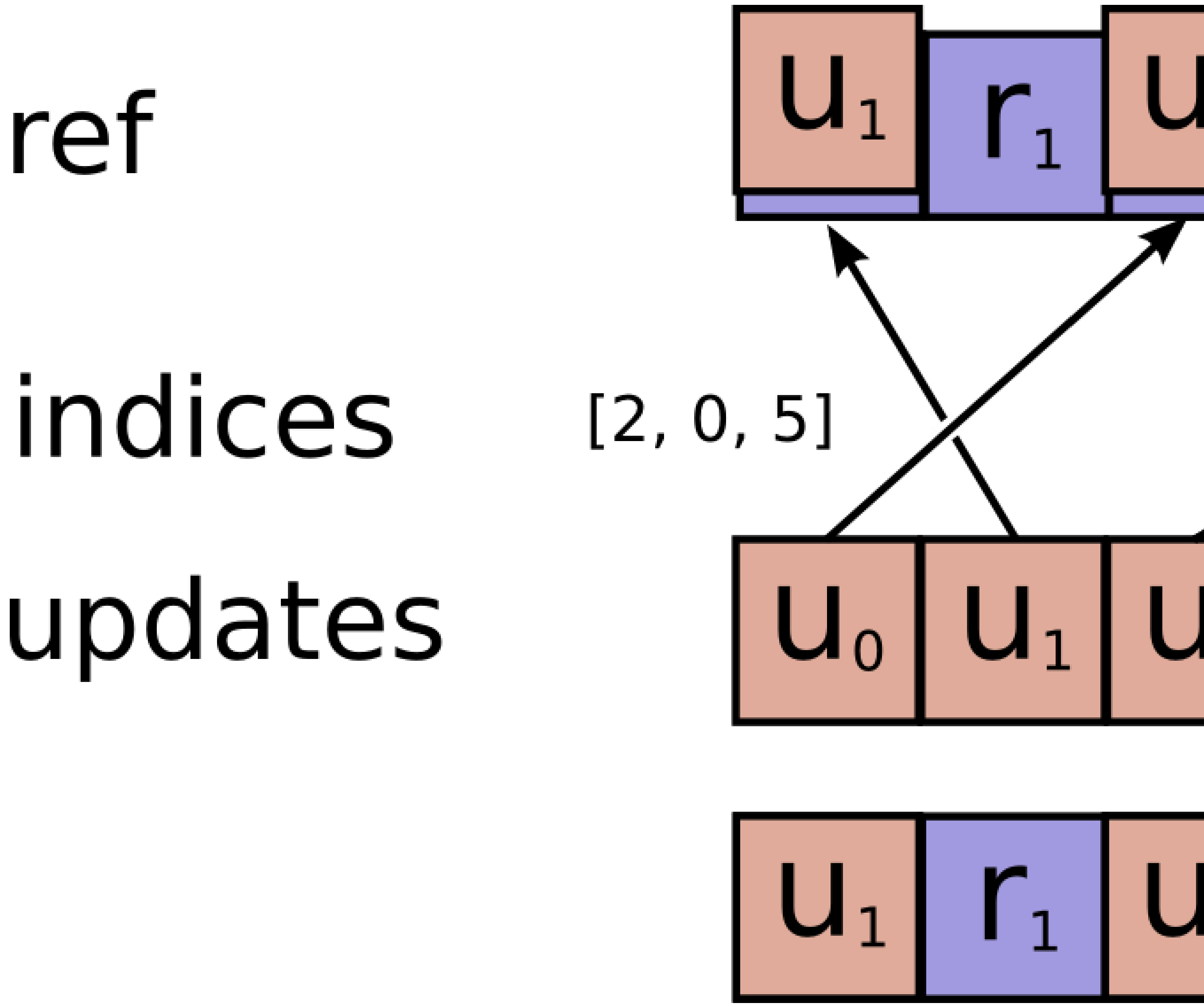
This operation computes

```
# Scalar indices
ref[indices, ...] = updates[...]

# Vector indices (for each i)
ref[indices[i], ...] = updates[i, ...]

# High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] = updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

If values in `ref` is to be updated more than once, because there are duplicate entires in `indices`, the order at which the updates happen for each value is undefined.

Requires `updates.shape = indices.shape + ref.shape[1:]`.



Args:

- `ref`: A mutable `Tensor`. Should be from a `Variable` node.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`.

  A tensor of indices into the first dimension of `ref`.
- `updates`: A `Tensor`. Must have the same type as `ref`. A tensor of updated values to store in `ref`.

- `use_locking`: An optional `bool`. Defaults to `True`. If True, the assignment will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
- `name`: A name for the operation (optional).

Returns:

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

`tf.scatter_add(ref, indices, updates, use_locking=None, name=None)`

Adds sparse updates to a variable reference.

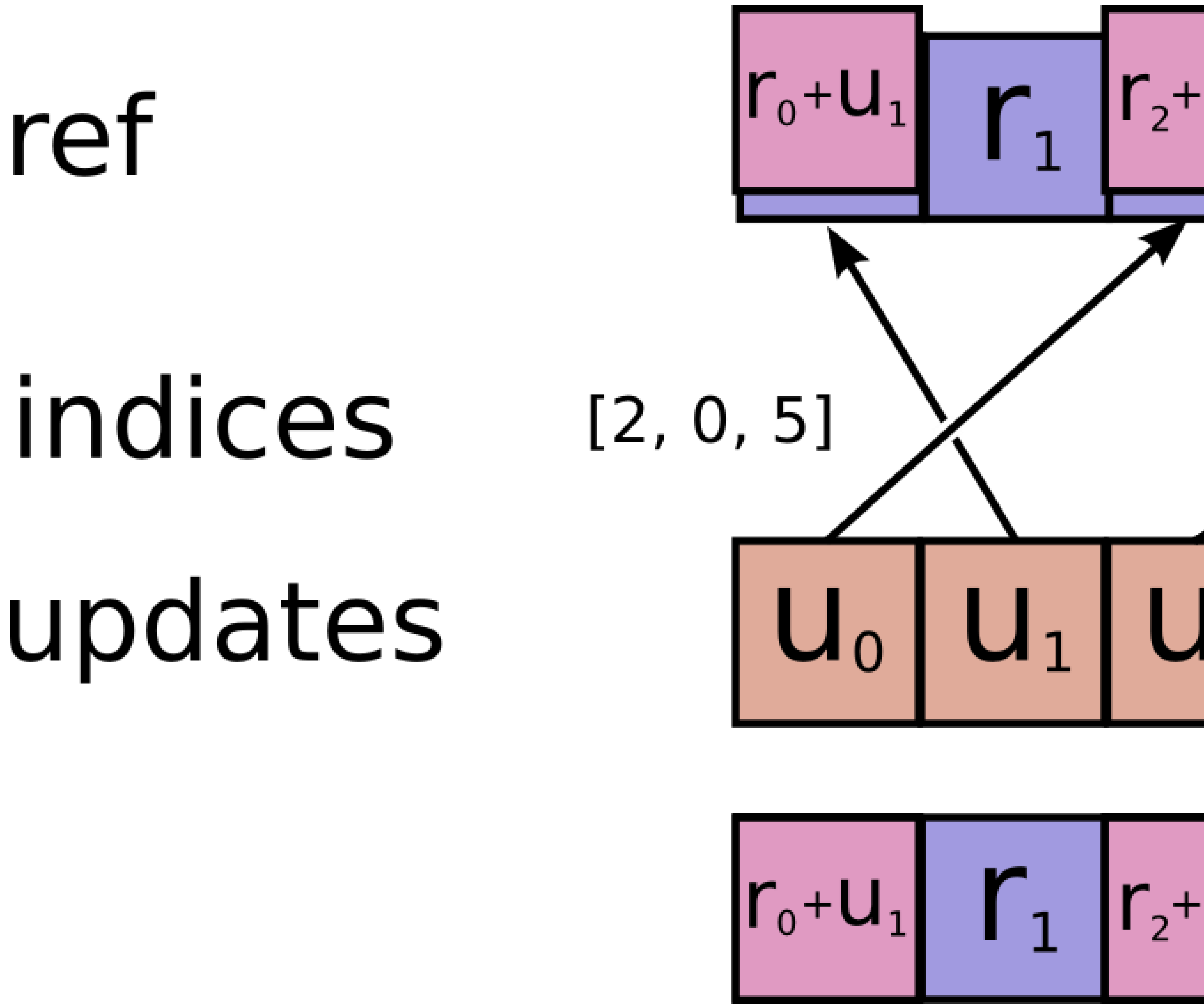
This operation computes

```
# Scalar indices
ref[indices, ...] += updates[...]

# Vector indices (for each i)
ref[indices[i], ...] += updates[i, ...]

# High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] += updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

Duplicate entries are handled correctly: if multiple `indices` reference the same location, their contributions add.

Requires `updates.shape = indices.shape + ref.shape[1:]`.



Args:

- `ref`: A mutable `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`,`uint16`, `int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`. Should be from a `Variable` node.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor of indices into the first dimension of `ref`.

- `updates`: A `Tensor`. Must have the same type as `ref`. A tensor of updated values to add to `ref`.
- `use_locking`: An optional `bool`. Defaults to `False`. If True, the addition will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
- `name`: A name for the operation (optional).

Returns:

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

`tf.scatter_sub(ref, indices, updates, use_locking=None, name=None)`

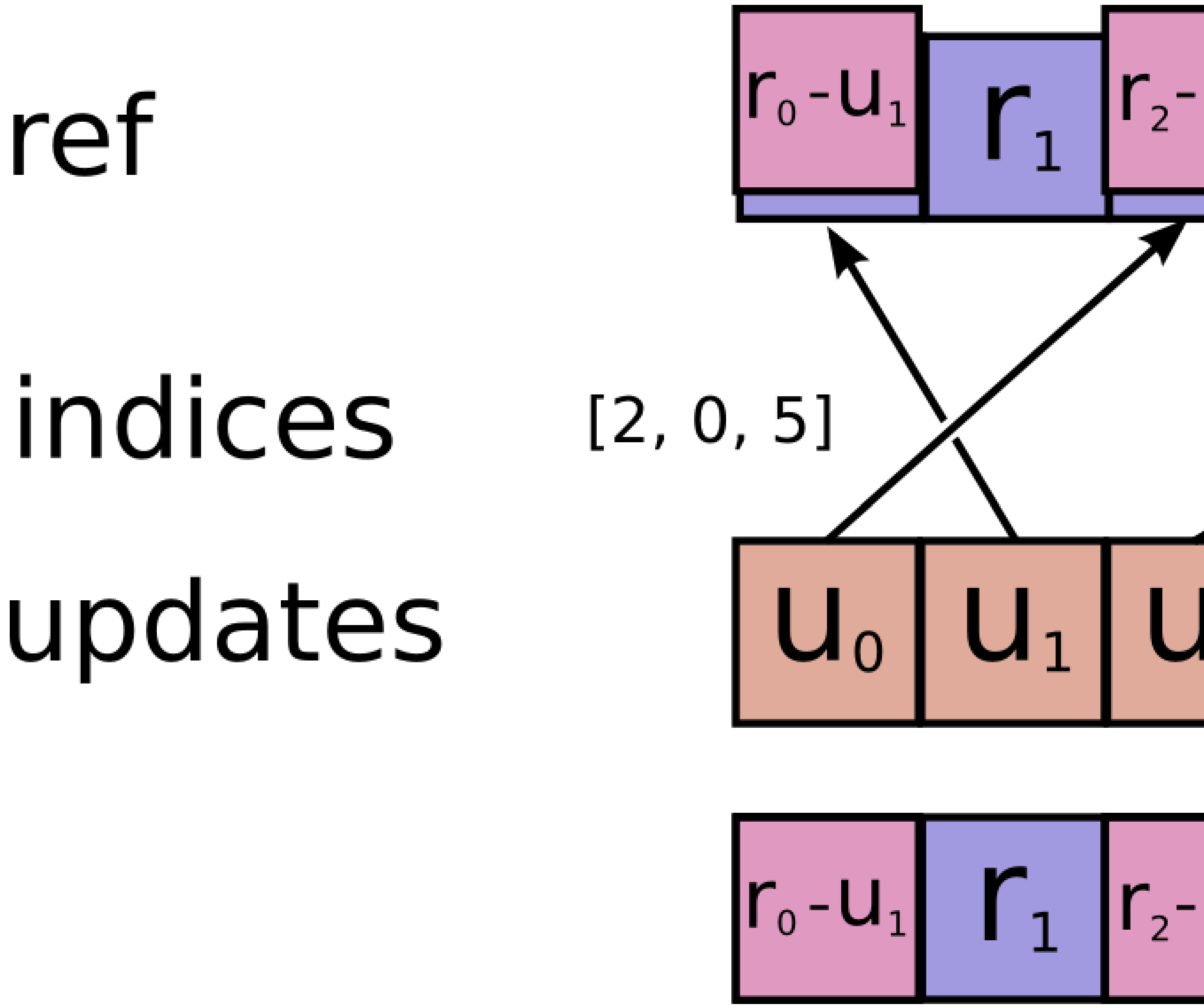
Subtracts sparse updates to a variable reference.

```
# Scalar indices
ref[indices, ...] -= updates[...]

# Vector indices (for each i)
ref[indices[i], ...] -= updates[i, ...]

# High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] -= updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

Duplicate entries are handled correctly: if multiple `indices` reference the same location, their (negated) contributions add.

Requires `updates.shape = indices.shape + ref.shape[1:]`.



Args:

- `ref`: A mutable `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`,`uint16`, `int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`. Should be from a `Variable` node.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor of indices into the first dimension of `ref`.

* `updates`: A `Tensor`. Must have the same type as `ref`. A tensor of updated values to subtract from `ref`.
* `use_locking`: An optional `bool`. Defaults to `False`. If True, the subtraction will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
* `name`: A name for the operation (optional).

Returns:

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

`tf.sparse_mask(a, mask_indices, name=None)`

Masks elements of `IndexedSlices`.

Given an `IndexedSlices` instance `a`, returns

another `IndexedSlices` that contains a subset of the slices of`a`. Only

the slices at indices specified in `mask_indices` are returned.
This is useful when you need to extract a subset of slices in an `IndexedSlices` object.

For example:

```
# `a` contains slices at indices [12, 26, 37, 45] from a large
tensor
# with shape [1000, 10]
a.indices => [12, 26, 37, 45]
tf.shape(a.values) => [4, 10]

# `b` will be the subset of `a` slices at its second and third
indices, so
```

```
# we want to mask of its first and last indices (which are at
absolute
# indices 12, 45)
b = tf.sparse_mask(a, [12, 45])

b.indices => [26, 37]
tf.shape(b.values) => [2, 10]
```

Args:

- `a`: An `IndexedSlices` instance.
- `mask_indices`: Indices of elements to mask.
- `name`: A name for the operation (optional).

Returns:

The masked `IndexedSlices` instance.

`class tf.IndexedSlices`

A sparse representation of a set of tensor slices at given indices.

This class is a simple wrapper for a pair of `Tensor` objects:

- `values`: A `Tensor` of any dtype with shape `[D0, D1, ..., Dn]`.
- `indices`: A 1-D integer `Tensor` with shape `[D0]`.

An `IndexedSlices` is typically used to represent a subset of a larger tensor `dense` of shape `[LARGE0, D1, .. , DN]` where `LARGE0 >>`

`D0`. The values in `indices` are the indices in the first dimension of the slices that have been extracted from the larger tensor.

The dense tensor `dense` represented by

an `IndexedSlices` `slices` has

```
dense[slices.indices[i], :, :, :, ...] =
slices.values[i, :, :, :, ...]
```

The `IndexedSlices` class is used principally in the definition of gradients for operations that have sparse gradients (e.g. `tf.gather`).

Contrast this representation with `SparseTensor`, which uses multi-dimensional indices and scalar values.

`tf.IndexedSlices.__init__(values, indices, dense_shape=None)`

Creates an `IndexedSlices`.

`tf.IndexedSlices.values`

A `Tensor` containing the values of the slices.

`tf.IndexedSlices.indices`

A 1-D `Tensor` containing the indices of the slices.

`tf.IndexedSlices.dense_shape`

A 1-D `Tensor` containing the shape of the corresponding dense tensor.

`tf.IndexedSlices.name`

The name of this `IndexedSlices`.

`tf.IndexedSlices.dtype`

The `DType` of elements in this tensor.

`tf.IndexedSlices.device`

The name of the device on which `values` will be produced, or `None`.

`tf.IndexedSlices.op`

The `Operation` that produces `values` as an output.

Other Methods

`tf.IndexedSlices.graph`

The `Graph` that contains the values, indices, and shape tensors.

Tensor Transformations

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

Contents

- `tf.gather(params, indices, validate_indices=None, name=None)`
- `tf.dynamic_partition(data, partitions, num_partitions, name=None)`
- `tf.dynamic_stitch(indices, data, name=None)`
- `tf.boolean_mask(tensor, mask, name=boolean_mask)`
- 2-D example
- Other Functions and Classes
- `tf.shape_n(input, name=None)`
- `tf.unique_with_counts(x, name=None)`

Casting

TensorFlow provides several operations that you can use to cast tensor data types in your graph.

`tf.string_to_number(string_tensor, out_type=None, name=None)`

Converts each string in the input Tensor to the specified numeric type.

(Note that int32 overflow results in an error while float overflow results in a rounded value.)

Args:

- `string_tensor`: A `Tensor` of type `string`.
- `out_type`: An optional `tf.DType` from: `tf.float32, tf.int32`.

  Defaults to `tf.float32`. The numeric type to interpret each string in string_tensor as.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `out_type`. A Tensor of the same shape as the input `string_tensor`.

`tf.to_double(x, name='ToDouble')`

Casts a tensor to type `float64`.

Args:

- `x`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `float64`.

Raises:

- `TypeError`: If `x` cannot be cast to the `float64`.

`tf.to_float(x, name='ToFloat')`

Casts a tensor to type `float32`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `float32`.

Raises:

* `TypeError`: If `x` cannot be cast to the `float32`.

`tf.to_bfloat16(x, name='ToBFloat16')`

Casts a tensor to type `bfloat16`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `bfloat16`.

Raises:

* `TypeError`: If `x` cannot be cast to the `bfloat16`.

```
tf.to_int32(x, name='ToInt32')
```

Casts a tensor to type `int32`.

Args:

- `x`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `int32`.

Raises:

- `TypeError`: If `x` cannot be cast to the `int32`.

```
tf.to_int64(x, name='ToInt64')
```

Casts a tensor to type `int64`.

Args:

- `x`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `int64`.

Raises:

- `TypeError`: If `x` cannot be cast to the `int64`.

`tf.cast(x, dtype, name=None)`

Casts a tensor to a new type.

The operation casts `x` (in case of `Tensor`) or `x.values` (in case of `SparseTensor`) to `dtype`.

For example:

```
# tensor `a` is [1.8, 2.2], dtype=tf.float
tf.cast(a, tf.int32) ==> [1, 2]  # dtype=tf.int32
```

Args:

- `x`: A `Tensor` or `SparseTensor`.
- `dtype`: The destination type.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x`.

Raises:

- `TypeError`: If `x` cannot be cast to the `dtype`.

Shapes and Shaping

TensorFlow provides several operations that you can use to determine the shape of a tensor and change the shape of a tensor.

`tf.shape(input, name=None)`

Returns the shape of a tensor.

This operation returns a 1-D integer tensor representing the shape of `input`.

For example:

```
# 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
shape(t) ==> [2, 2, 3]
```

Args:

- `input`: A `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

`tf.size(input, name=None)`

Returns the size of a tensor.

This operation returns an integer representing the number of elements in `input`.

For example:

```
# 't' is [[[1, 1,, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]]
size(t) ==> 12
```

Args:

- `input`: A `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

`tf.rank(input, name=None)`

Returns the rank of a tensor.

This operation returns an integer representing the rank of `input`.

For example:

```
# 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
# shape of tensor 't' is [2, 2, 3]
rank(t) ==> 3
```

Note: The rank of a tensor is not the same as the rank of a matrix. The rank of a tensor is the number of indices required to uniquely select each element of the tensor. Rank is also known as "order", "degree", or "ndims."

Args:

- `input`: A `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

```
tf.reshape(tensor, shape, name=None)
```

Reshapes a tensor.

Given `tensor`, this operation returns a tensor that has the same values as `tensor` with shape `shape`.

If one component of `shape` is the special value -1, the size of that dimension is computed so that the total size remains constant. In particular, a `shape` of `[-1]` flattens into 1-D. At most one component of `shape` can be -1.

If `shape` is 1-D or higher, then the operation returns a tensor with shape `shape` filled with the values of `tensor`. In this case, the number of elements implied by `shape` must be the same as the number of elements in `tensor`.

For example:

```
# tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
# tensor 't' has shape [9]
reshape(t, [3, 3]) ==> [[1, 2, 3]
                    [4, 5, 6]
                    [7, 8, 9]]

# tensor 't' is [[[1, 1], [2, 2]]
#               [[3, 3], [4, 4]]]
# tensor 't' has shape [2, 2, 2]
reshape(t, [2, 4]) ==> [[1, 1, 2, 2]
                    [3, 3, 4, 4]]

# tensor 't' is [[[1, 1, 1],
#                [2, 2, 2]],
#               [[3, 3, 3],
#                [4, 4, 4]],
#               [[5, 5, 5],
#                [6, 6, 6]]]
# tensor 't' has shape [3, 2, 3]
# pass '[-1]' to flatten 't'
reshape(t, [-1]) ==> [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5,
5, 6, 6, 6]
# -1 can also be used with higher dimensional shapes
reshape(t, [2, -1]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],
                     [4, 4, 4, 5, 5, 5, 6, 6, 6]]

# tensor 't' is [7]
# shape `[]` reshapes to a scalar
reshape(t, []) ==> 7
```

Args:

- `tensor`: A `Tensor`.
- `shape`: A `Tensor` of type `int32`. Defines the shape of the output tensor.
- `name`: A name for the operation (optional).

Returns:

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

`tf.squeeze(input, squeeze_dims=None, name=None)`

Removes dimensions of size 1 from the shape of a tensor.

Given a tensor `input`, this operation returns a tensor of the same type with all dimensions of size 1 removed. If you don't want to remove all size 1 dimensions, you can remove specific size 1 dimensions by specifying`squeeze_dims`.

For example:

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t)) ==> [2, 3]
```

Or, to remove specific size 1 dimensions:

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

Args:

- `input`: A `Tensor`. The `input` to squeeze.
- `squeeze_dims`: An optional list of `ints`. Defaults to `[]`. If specified, only squeezes the dimensions listed. The dimension index starts at 0. It is an error to squeeze a dimension that is not 1.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Contains the same data as `input`, but has one or more dimensions of size 1 removed.

`tf.expand_dims(input, dim, name=None)`

Inserts a dimension of 1 into a tensor's shape.

Given a tensor `input`, this operation inserts a dimension of 1 at the dimension index `dim` of `input`'s shape. The dimension index `dim` starts at zero; if you specify a negative number for `dim` it is counted backward from the end.
This operation is useful if you want to add a batch dimension to a single element. For example, if you have a single image of shape `[height, width, channels]`, you can make it a batch of 1 image with`expand_dims(image, 0)`, which will make the shape `[1, height, width, channels]`.

Other examples:

```
# 't' is a tensor of shape [2]
shape(expand_dims(t, 0)) ==> [1, 2]
shape(expand_dims(t, 1)) ==> [2, 1]
shape(expand_dims(t, -1)) ==> [2, 1]

# 't2' is a tensor of shape [2, 3, 5]
shape(expand_dims(t2, 0)) ==> [1, 2, 3, 5]
shape(expand_dims(t2, 2)) ==> [2, 3, 1, 5]
shape(expand_dims(t2, 3)) ==> [2, 3, 5, 1]
```

This operation requires that:

`-1-input.dims() <= dim <= input.dims()`

This operation is related to `squeeze()`, which removes dimensions of size 1.

Args:

- `input`: A `Tensor`.
- `dim`: A `Tensor` of type `int32`. 0-D (scalar). Specifies the dimension index at which to expand the shape of`input`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Contains the same data as `input`, but its shape has an additional dimension of size 1 added.

Slicing and Joining

TensorFlow provides several operations to slice or extract parts of a tensor, or join multiple tensors together.

`tf.slice(input_, begin, size, name=None)`

Extracts a slice from a tensor.

This operation extracts a slice of size `size` from a tensor `input` starting at the location specified by `begin`. The slice `size` is represented as a tensor shape, where `size[i]` is the number of elements of the 'i'th dimension of`input` that you want to slice. The starting location (`begin`) for the slice is represented as an offset in each dimension of `input`. In other words, `begin[i]` is the offset into the 'i'th dimension of `input` that you want to slice from.

`begin` is zero-based; `size` is one-based. If `size[i]` is -1, all remaining elements in dimension i are included in the slice. In other words, this is equivalent to setting:
`size[i] = input.dim_size(i) - begin[i]`

This operation requires that:

`0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n]`

For example:

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

Args:

- `input_`: A `Tensor`.
- `begin`: An `int32` or `int64` `Tensor`.
- `size`: An `int32` or `int64` `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` the same type as `input`.

`tf.split(split_dim, num_split, value, name='split')`

Splits a tensor into `num_split` tensors along one dimension.

Splits `value` along dimension `split_dim` into `num_split` smaller tensors. Requires that `num_split` evenly divide `value.shape[split_dim]`.

For example:

```
# 'value' is a tensor with shape [5, 30]
# Split 'value' into 3 tensors along dimension 1
split0, split1, split2 = tf.split(1, 3, value)
tf.shape(split0) ==> [5, 10]
```

Args:

- `split_dim`: A 0-D `int32` `Tensor`. The dimension along which to split. Must be in the range `[0, rank(value))`.
- `num_split`: A Python integer. The number of ways to split.
- `value`: The `Tensor` to split.
- `name`: A name for the operation (optional).

Returns:

`num_split` `Tensor` objects resulting from splitting `value`.

`tf.tile(input, multiples, name=None)`

Constructs a tensor by tiling a given tensor.

This operation creates a new tensor by replicating `input` `multiples` times. The output tensor's i'th dimension has `input.dims(i) * multiples[i]` elements, and the values of `input` are replicated `multiples[i]` times along the 'i'th dimension. For example, tiling `[a b c d]` by `[2]` produces `[a b c d a b c d]`.

Args:

- `input`: A `Tensor`. 1-D or higher.
- `multiples`: A `Tensor` of type `int32`. 1-D. Length must be the same as the number of dimensions in `input`
- `name`: A name for the operation (optional).

Returns:

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

`tf.pad(input, paddings, name=None)`

Pads a tensor with zeros.

This operation pads a `input` with zeros according to the `paddings` you specify. `paddings` is an integer tensor with shape `[Dn, 2]`, where n is the rank of `input`. For each dimension D of `input`, `paddings[D, 0]`indicates how many zeros to add before the contents of `input` in that dimension, and `paddings[D, 1]`indicates how many zeros to add after the contents of `input` in that dimension.

The padded size of each dimension D of the output is:

`paddings(D, 0) + input.dim_size(D) + paddings(D, 1)`

For example:

```
# 't' is [[1, 1], [2, 2]]
# 'paddings' is [[1, 1], [2, 2]]
# rank of 't' is 2
pad(t, paddings) ==> [[0, 0, 0, 0, 0, 0]
                      [0, 0, 1, 1, 0, 0]
                      [0, 0, 2, 2, 0, 0]
                      [0, 0, 0, 0, 0, 0]]
```

Args:

- `input`: A `Tensor`.
- `paddings`: A `Tensor` of type `int32`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.concat(concat_dim, values, name='concat')`

Concatenates tensors along one dimension.

Concatenates the list of tensors `values` along dimension `concat_dim`. If `values[i].shape = [D0, D1, ... Dconcat_dim(i), ...Dn]`, the concatenated result has shape

```
[D0, D1, ... Rconcat_dim, ...Dn]
```

where

```
Rconcat_dim = sum(Dconcat_dim(i))
```

That is, the data from the input tensors is joined along the `concat_dim` dimension.
The number of dimensions of the input tensors must match, and all dimensions except `concat_dim` must be equal.

For example:

```
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
tf.concat(0, [t1, t2]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10,
11, 12]]
tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11,
12]]

# tensor t3 with shape [2, 3]
# tensor t4 with shape [2, 3]
tf.shape(tf.concat(0, [t3, t4])) ==> [4, 3]
tf.shape(tf.concat(1, [t3, t4])) ==> [2, 6]
```

Args:

- `concat_dim`: 0-D `int32` `Tensor`. Dimension along which to concatenate.

- `values`: A list of `Tensor` objects or a single `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` resulting from concatenation of the input tensors.

`tf.pack(values, name='pack')`

Packs a list of rank-`R` tensors into one rank-`(R+1)` tensor.

Packs tensors in `values` into a tensor with rank one higher than each tensor in `values` and shape `[len(values)] + values[0].shape`. The output satisfies `output[i, ...] = values[i][...]`.

This is the opposite of unpack. The numpy equivalent is

```
tf.pack([x, y, z]) = np.asarray([x, y, z])
```

Args:

- `values`: A list of `Tensor` objects with the same shape and type.
- `name`: A name for this operation (optional).

Returns:

- `output`: A packed `Tensor` with the same type as `values`.

`tf.unpack(value, num=None, name='unpack')`

Unpacks the outer dimension of a rank-`R` tensor into rank-`(R-1)` tensors.

Unpacks `num` tensors from `value` along the first dimension. If `num` is not specified (the default), it is inferred from`value`'s shape.

If `value.shape[0]` is not known, `ValueError` is raised.

The ith tensor in `output` is the slice `value[i, ...]`. Each tensor in `output` has shape `value.shape[1:]`.

This is the opposite of pack. The numpy equivalent is

```
tf.unpack(x, n) = list(x)
```

Args:

- `value`: A rank `R > 0` `Tensor` to be unpacked.
- `num`: An `int`. The first dimension of value. Automatically inferred if `None` (the default).
- `name`: A name for the operation (optional).

Returns:

The list of `Tensor` objects unpacked from `value`.

Raises:

- `ValueError`: If `num` is unspecified and cannot be inferred.

`tf.reverse_sequence(input, seq_lengths, seq_dim, batch_dim=None, name=None)`

Reverses variable length slices.

This op first slices `input` along the dimension `batch_dim`, and for each slice `i`, reverses the first`seq_lengths[i]` elements along the dimension `seq_dim`.

The elements of `seq_lengths` must obey `seq_lengths[i] < input.dims[seq_dim]`, and`seq_lengths` must be a vector of length `input.dims[batch_dim]`.

The output slice `i` along dimension `batch_dim` is then given by input slice `i`, with the first `seq_lengths[i]`slices along dimension `seq_dim` reversed.

For example:

```
# Given this:
batch_dim = 0
seq_dim = 1
input.dims = (4, 8, ...)
seq_lengths = [7, 2, 3, 5]

# then slices of input are reversed on seq_dim, but only up to
seq_lengths:
output[0, 0:7, :, ...] = input[0, 7:0:-1, :, ...]
output[1, 0:2, :, ...] = input[1, 2:0:-1, :, ...]
output[2, 0:3, :, ...] = input[2, 3:0:-1, :, ...]
output[3, 0:5, :, ...] = input[3, 5:0:-1, :, ...]

# while entries past seq_lens are copied through:
output[0, 7:, :, ...] = input[0, 7:, :, ...]
output[1, 2:, :, ...] = input[1, 2:, :, ...]
```

```
output[2, 3:, :, ...] = input[2, 3:, :, ...]
output[3, 2:, :, ...] = input[3, 2:, :, ...]
```

In contrast, if:

```
# Given this:
batch_dim = 2
seq_dim = 0
input.dims = (8, ?, 4, ...)
seq_lengths = [7, 2, 3, 5]

# then slices of input are reversed on seq_dim, but only up to
seq_lengths:
output[0:7, :, 0, :, ...] = input[7:0:-1, :, 0, :, ...]
output[0:2, :, 1, :, ...] = input[2:0:-1, :, 1, :, ...]
output[0:3, :, 2, :, ...] = input[3:0:-1, :, 2, :, ...]
output[0:5, :, 3, :, ...] = input[5:0:-1, :, 3, :, ...]

# while entries past seq_lens are copied through:
output[7:, :, 0, :, ...] = input[7:, :, 0, :, ...]
output[2:, :, 1, :, ...] = input[2:, :, 1, :, ...]
output[3:, :, 2, :, ...] = input[3:, :, 2, :, ...]
output[2:, :, 3, :, ...] = input[2:, :, 3, :, ...]
```

Args:

- `input`: A `Tensor`. The input to reverse.
- `seq_lengths`: A `Tensor` of type `int64`. 1-D with length `input.dims(batch_dim)` and`max(seq_lengths) < input.dims(seq_dim)`
- `seq_dim`: An `int`. The dimension which is partially reversed.
- `batch_dim`: An optional `int`. Defaults to `0`. The dimension along which reversal is performed.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. The partially reversed input. It has the same shape as `input`.

`tf.reverse(tensor, dims, name=None)`

Reverses specific dimensions of a tensor.

Given a `tensor`, and a `bool` tensor `dims` representing the dimensions of `tensor`, this operation reverses each dimension i of `tensor` where `dims[i]` is `True`.

`tensor` can have up to 8 dimensions. The number of dimensions of `tensor` must equal the number of elements in `dims`. In other words:
`rank(tensor) = size(dims)`

For example:

```
# tensor 't' is [[[[ 0,  1,  2,  3],
#                  [ 4,  5,  6,  7],
#                  [ 8,  9, 10, 11]],
#                 [[12, 13, 14, 15],
#                  [16, 17, 18, 19],
#                  [20, 21, 22, 23]]]]
# tensor 't' shape is [1, 2, 3, 4]

# 'dims' is [False, False, False, True]
reverse(t, dims) ==> [[[[ 3,  2,  1,  0],
                        [ 7,  6,  5,  4],
                        [ 11, 10, 9, 8]],
                       [[15, 14, 13, 12],
                        [19, 18, 17, 16],
                        [23, 22, 21, 20]]]]

# 'dims' is [False, True, False, False]
```

```
reverse(t, dims) ==> [[[[12, 13, 14, 15],
                        [16, 17, 18, 19],
                        [20, 21, 22, 23]
                       [[ 0,  1,  2,  3],
                        [ 4,  5,  6,  7],
                        [ 8,  9, 10, 11]]]]

# 'dims' is [False, False, True, False]
reverse(t, dims) ==> [[[[8, 9, 10, 11],
                        [4, 5, 6, 7],
                        [0, 1, 2, 3]]
                       [[20, 21, 22, 23],
                        [16, 17, 18, 19],
                        [12, 13, 14, 15]]]]
```

Args:

- `tensor`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int32`, `bool`, `float32`, `float64`. Up to 8-D.
- `dims`: A `Tensor` of type `bool`. 1-D. The dimensions to reverse.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `tensor`. The same shape as `tensor`.

`tf.transpose(a, perm=None, name='transpose')`

Transposes `a`. Permutes the dimensions according to `perm`.
The returned tensor's dimension i will correspond to the input dimension `perm[i]`. If `perm` is not given, it is set to (n-1...0), where n

is the rank of the input tensor. Hence by default, this operation performs a regular matrix transpose on 2-D input Tensors.

For example:

```
# 'x' is [[1 2 3]
#         [4 5 6]]
tf.transpose(x) ==> [[1 4]
                     [2 5]
                     [3 6]]

# Equivalently
tf.transpose(x, perm=[1, 0]) ==> [[1 4]
                                  [2 5]
                                  [3 6]]

# 'perm' is more useful for n-dimensional tensors, for n > 2
# 'x' is   [[[1  2  3]
#            [4  5  6]]
#           [[7  8  9]
#            [10 11 12]]]
# Take the transpose of the matrices in dimension-0
tf.transpose(b, perm=[0, 2, 1]) ==> [[[1  4]
                                      [2  5]
                                      [3  6]]

                                     [[7 10]
                                      [8 11]
                                      [9 12]]]
```

Args:

- `a`: A `Tensor`.
- `perm`: A permutation of the dimensions of `a`.
- `name`: A name for the operation (optional).

Returns:

A transposed `Tensor`.

`tf.space_to_depth(input, block_size, name=None)`

SpaceToDepth for tensors of type T.

Rearranges blocks of spatial data, into depth. More specifically, this op outputs a copy of the input tensor where values from the `height` and `width` dimensions are moved to the `depth` dimension.

The attr `block_size`indicates the input block size and how the data is moved.

- Non-overlapping blocks of size `block_size x block size` are rearranged into depth at each location.
- The depth of the output tensor is `input_depth * block_size * block_size`.
- The input tensor's height and width must be divisible by block_size.

That is, assuming the input is in the shape: `[batch, height, width, depth]`, the shape of the output will be: `[batch, height/block_size, width/block_size, depth*block_size*block_size]`

This operation requires that the input tensor be of rank 4, and that `block_size` be >=1 and a divisor of both the input `height` and `width`.

This operation is useful for resizing the activations between convolutions (but keeping all data), e.g. instead of pooling. It is also useful for training purely convolutional models.

For example, given this input of shape `[1, 2, 2, 1]`, and block_size of 2:

```
x = [[[[1], [2]],
      [[3], [4]]]]
```

This operation will output a tensor of shape `[1, 1, 1, 4]`:

```
[[[[1, 2, 3, 4]]]]
```

Here, the input has a batch of 1 and each batch element has shape `[2, 2, 1]`, the corresponding output will have a single element (i.e. width and height are both 1) and will have a depth of 4 channels (1 * block_size * block_size). The output element shape is `[1, 1, 4]`.

For an input tensor with larger depth, here of shape `[1, 2, 2, 3]`, e.g.

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

This operation, for block_size of 2, will return the following tensor of shape `[1, 1, 1, 12]`

```
[[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

Similarly, for the following input of shape `[1 4 4 1]`, and a block size of 2:

```
x = [[ [1],   [2],  [5],  [6]],
     [ [3],   [4],  [7],  [8]],
     [ [9],  [10], [13],  [14]],
     [ [11], [12], [15],  [16]]]
```

the operator will return the following tensor of shape `[1 2 2 4]`:

```
x = [[[[1, 2, 3, 4],
       [5, 6, 7, 8]],
      [[9, 10, 11, 12],
       [13, 14, 15, 16]]]]
```

Args:

- `input`: A `Tensor`.
- `block_size`: An `int`. The size of the spatial block.
- `name`: A name for the operation (optional).

Returns:

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

```
tf.depth_to_space(input, block_size, name=None)
```

DepthToSpace for tensors of type T.

Rearranges data from depth into blocks of spatial data. This is the reverse transformation of SpaceToDepth. More specifically, this op outputs a copy of the input tensor where values from

the `depth` dimension are moved in spatial blocks to

the `height` and `width` dimensions. The attr `block_size` indicates the input block size and how the data is moved.

- Chunks of data of size `block_size * block_size` from depth are rearranged into non-overlapping blocks of size `block_size x block_size`
- The width the output tensor is `input_depth * block_size`, whereas the height is `input_height * block_size`.
- The depth of the input tensor must be divisible by `block_size * block_size`.

That is, assuming the input is in the shape: `[batch, height, width, depth]`, the shape of the output will be: `[batch, height*block_size, width*block_size, depth/(block_size*block_size)]`

This operation requires that the input tensor be of rank 4, and that `block_size` be >=1 and that `block_size * block_size` be a divisor of the input depth.

This operation is useful for resizing the activations between convolutions (but keeping all data), e.g. instead of pooling. It is also useful for training purely convolutional models.

For example, given this input of shape `[1, 1, 1, 4]`, and a block size of 2:

```
x = [[[[1, 2, 3, 4]]]]
```

This operation will output a tensor of shape `[1, 2, 2, 1]`:

```
   [[[[1], [2]],
     [[3], [4]]]]
```

Here, the input has a batch of 1 and each batch element has shape `[1, 1, 4]`, the corresponding output will have 2x2 elements and will have a depth of 1 channel (1 = `4 / (block_size * block_size)`). The output element shape is `[2, 2, 1]`.

For an input tensor with larger depth, here of shape `[1, 1, 1, 12]`, e.g.

```
x = [[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

This operation, for block size of 2, will return the following tensor of shape `[1, 2, 2, 3]`

```
   [[[[1, 2, 3], [4, 5, 6]],
     [[7, 8, 9], [10, 11, 12]]]]
```

Similarly, for the following input of shape `[1 2 2 4]`, and a block size of 2:

```
x =  [[[[1, 2, 3, 4],
       [5, 6, 7, 8]],
      [[9, 10, 11, 12],
       [13, 14, 15, 16]]]]
```

the operator will return the following tensor of shape `[1 4 4 1]`:

```
x = [[ [1],   [2],  [5],  [6]],
     [ [3],   [4],  [7],  [8]],
     [ [9],  [10], [13],  [14]],
     [ [11], [12], [15],  [16]]]
```

Args:

* `input`: A `Tensor`.
* `block_size`: An `int`. The size of the spatial block, same as in Space2Depth.
* `name`: A name for the operation (optional).

Returns:

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

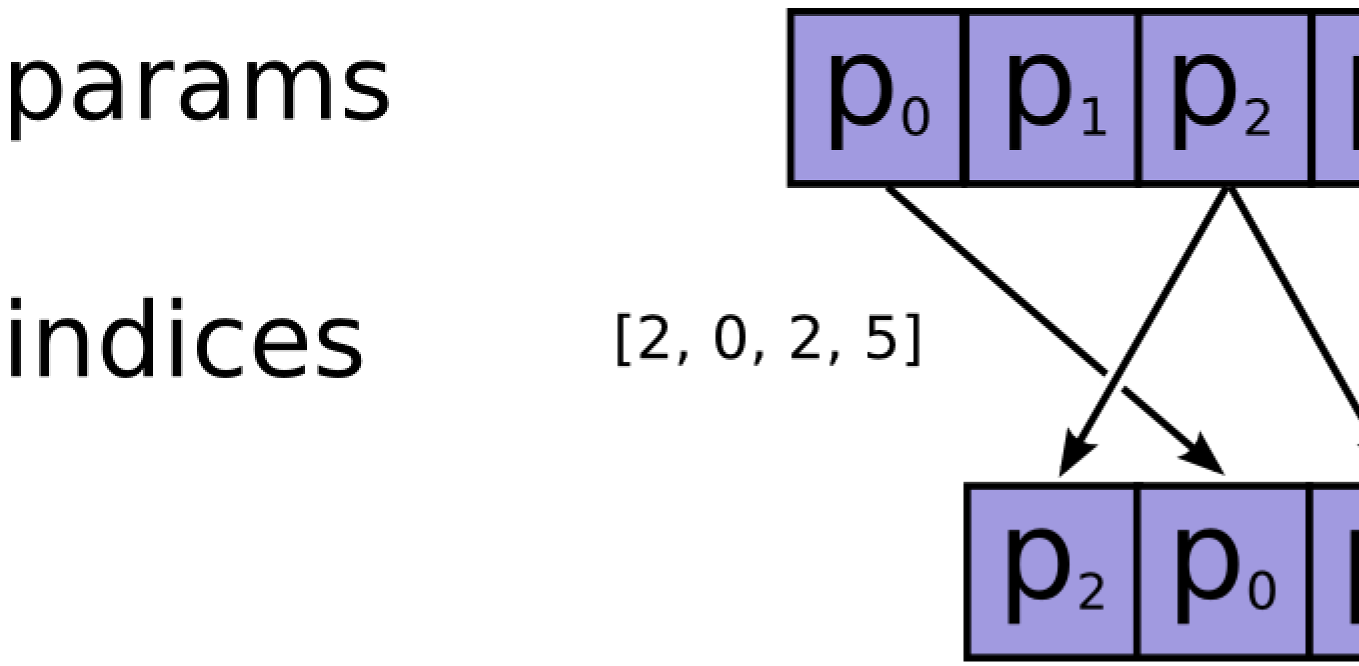
`tf.gather(params, indices, validate_indices=None, name=None)`

Gather slices from `params` according to `indices`.

`indices` must be an integer tensor of any dimension (usually 0-D or 1-D). Produces an output tensor with shape`indices.shape + params.shape[1:]` where:

```
# Scalar indices
output[:, ..., :] = params[indices, :, ... :]

# Vector indices
output[i, :, ..., :] = params[indices[i], :, ... :]

# Higher rank indices
output[i, ..., j, :, ... :] = params[indices[i, ...,
j], :, ..., :]
```

If `indices` is a permutation and `len(indices) == params.shape[0]` then this operation will permute`params` accordingly.



Args:

- `params`: A `Tensor`.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- `validate_indices`: An optional `bool`. Defaults to `True`.
- `name`: A name for the operation (optional).

Returns:

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

```
tf.dynamic_partition(data, partitions, num_partitions,
name=None)
```

Partitions `data` into `num_partitions` tensors using indices

from `partitions`.

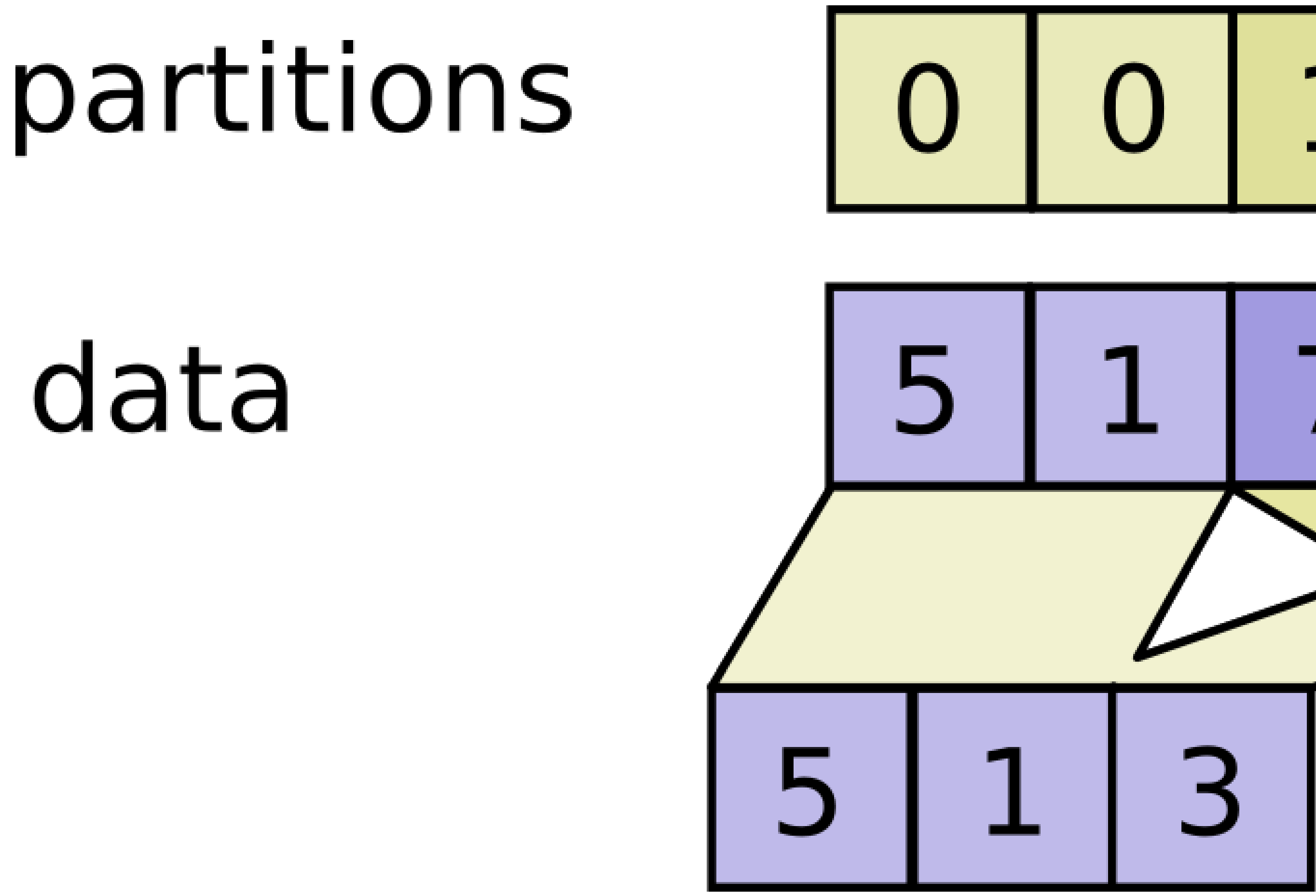
For each index tuple `js` of size `partitions.ndim`, the slice `data[js, ...]` becomes part of`outputs[partitions[js]]`. The slices with `partitions[js] = i` are placed in `outputs[i]` in lexicographic order of `js`, and the first dimension of `outputs[i]` is the number of entries in `partitions` equal to `i`. In detail,

```
outputs[i].shape = [sum(partitions == i)] +
data.shape[partitions.ndim:]

outputs[i] = pack([data[js, ...] for js if partitions[js] == i])
```

`data.shape` must start with `partitions.shape`.

For example:

```
# Scalar partitions
partitions = 1
num_partitions = 2
data = [10, 20]
outputs[0] = []  # Empty with shape [0, 2]
outputs[1] = [[10, 20]]

# Vector partitions
partitions = [0, 0, 1, 1, 0]
num_partitions = 2
data = [10, 20, 30, 40, 50]
outputs[0] = [10, 20, 50]
outputs[1] = [30, 40]
```



Args:

- `data`: A `Tensor`.
- `partitions`: A `Tensor` of type `int32`. Any shape. Indices in the range `[0, num_partitions)`.
- `num_partitions`: An `int` that is `>= 1`. The number of partitions to output.
- `name`: A name for the operation (optional).

Returns:

A list of `num_partitions` `Tensor` objects of the same type as data.

`tf.dynamic_stitch(indices, data, name=None)`

Interleave the values from the `data` tensors into a single tensor.

Builds a merged tensor such that

```
merged[indices[m][i, ..., j], ...] = data[m][i, ..., j, ...]
```

For example, if each `indices[m]` is scalar or vector, we have

```
# Scalar indices
merged[indices[m], ...] = data[m][...]

# Vector indices
merged[indices[m][i], ...] = data[m][i, ...]
```

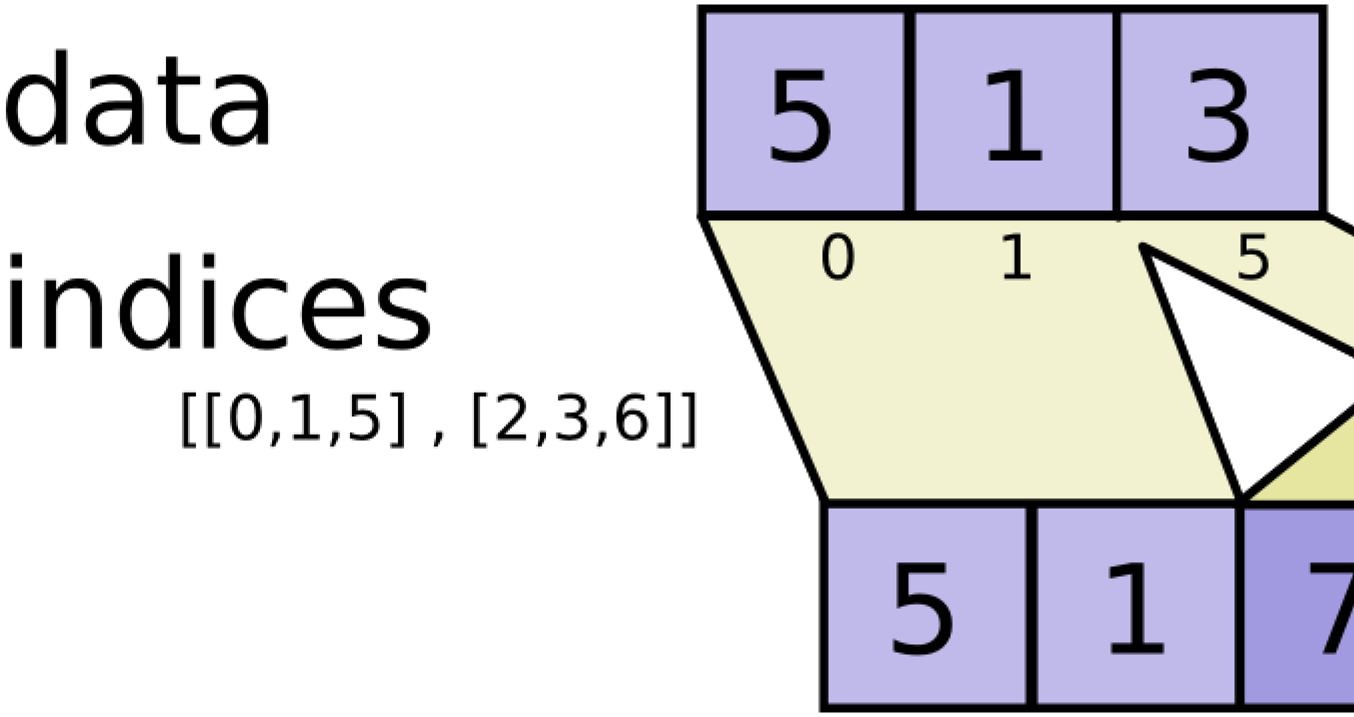
Each `data[i].shape` must start with the corresponding `indices[i].shape`, and the rest of`data[i].shape` must be constant w.r.t. `i`. That is, we must have `data[i].shape = indices[i].shape + constant`. In terms of this `constant`, the output shape is

```
merged.shape = [max(indices)] + constant
```

Values are merged in order, so if an index appears in both `indices[m][i]` and `indices[n][j]` for `(m,i) < (n,j)` the slice `data[n][j]` will appear in the merged result.

For example:

```
indices[0] = 6
indices[1] = [4, 1]
indices[2] = [[5, 2], [0, 3]]
data[0] = [61, 62]
data[1] = [[41, 42], [11, 12]]
data[2] = [[[51, 52], [21, 22]], [[1, 2], [31, 32]]]
merged = [[1, 2], [11, 12], [21, 22], [31, 32], [41, 42],
          [51, 52], [61, 62]]
```



Args:

- `indices`: A list of at least 2 `Tensor` objects of type `int32`.
- `data`: A list with the same number of `Tensor` objects

  as `indices` of `Tensor` objects of the same type.
- `name`: A name for the operation (optional).

Returns:

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

`tf.boolean_mask(tensor, mask, name='boolean_mask')`

Apply boolean mask to tensor. Numpy equivalent is `tensor[mask]`.

```
# 1-D example
```

```
tensor = [0, 1, 2, 3]
mask = [True, False, True, False]
boolean_mask(tensor, mask) ==> [0, 2]
```

In general, `0 < dim(mask) = K <= dim(tensor)`, and `mask`'s shape must match the first K dimensions of`tensor`'s shape. We then have: `boolean_mask(tensor, mask)[i, j1,...,jd] = tensor[i1,...,iK,j1,...,jd]` where `(i1,...,iK)` is the ith `True` entry of `mask` (row-major order).

Args:

- `tensor`: N-D tensor. First K dimensions can be None, which allows e.g. undefined batch size. Trailing dimensions must be specified.
- `mask`: K-D boolean tensor, K <= N.
- `name`: A name for this operation (optional).

Returns:

Tensor populated by entries in `tensor` corresponding to `True` values in `mask`.

Raises:

- `ValueError`: If shapes do not conform.
- `Examples`: ```python

2-D example

a = [[1, 2], [3, 4], [5, 6]] mask = [True, False, True]
boolean_mask(tensor, mask) ==> [[1, 2], [5, 6]] ```

Other Functions and Classes

`tf.shape_n(input, name=None)`

Returns shape of tensors.

This operation returns N 1-D integer tensors representing shape of `input[i]s`.

Args:

- `input`: A list of at least 1 `Tensor` objects of the same type.
- `name`: A name for the operation (optional).

Returns:

A list with the same number of `Tensor` objects as `input` of `Tensor` objects of type `int32`.

`tf.unique_with_counts(x, name=None)`

Finds unique elements in a 1-D tensor.

This operation returns a tensor `y` containing all of the unique elements of `x` sorted in the same order that they occur in `x`. This operation also returns a tensor `idx` the same size as `x` that contains the index of each value of `x`in the unique output `y`. Finally, it returns a third tensor `count` that contains the count of each element of `y` in `x`. In other words:

`y[idx[i]] = x[i] for i in [0, 1,...,rank(x) - 1]`

For example:

```
# tensor 'x' is [1, 1, 2, 4, 4, 4, 7, 8, 8]
y, idx, count = unique_with_counts(x)
y ==> [1, 2, 4, 7, 8]
idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
count ==> [2, 1, 3, 1, 2]
```

Args:

- `x`: A `Tensor`. 1-D.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (y, idx, count).

- `y`: A `Tensor`. Has the same type as `x`. 1-D.
- `idx`: A `Tensor` of type `int32`. 1-D.
- `count`: A `Tensor` of type `int32`. 1-D.

Math

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

Contents

- Math
- Arithmetic Operators
- `tf.add(x, y, name=None)`
- `tf.sub(x, y, name=None)`
- `tf.mul(x, y, name=None)`
- `tf.div(x, y, name=None)`
- `tf.truediv(x, y, name=None)`
- `tf.floordiv(x, y, name=None)`
- `tf.mod(x, y, name=None)`
- `tf.cross(a, b, name=None)`
- Basic Math Functions
- `tf.add_n(inputs, name=None)`
- `tf.abs(x, name=None)`
- `tf.neg(x, name=None)`
- `tf.sign(x, name=None)`
- `tf.inv(x, name=None)`
- `tf.square(x, name=None)`
- `tf.round(x, name=None)`
- `tf.sqrt(x, name=None)`
- `tf.rsqrt(x, name=None)`
- `tf.pow(x, y, name=None)`
- `tf.exp(x, name=None)`
- `tf.log(x, name=None)`
- `tf.ceil(x, name=None)`
- `tf.floor(x, name=None)`
- `tf.maximum(x, y, name=None)`
- `tf.minimum(x, y, name=None)`
- `tf.cos(x, name=None)`
- `tf.sin(x, name=None)`
- `tf.lgamma(x, name=None)`
- `tf.erf(x, name=None)`
- `tf.erfc(x, name=None)`
- Matrix Math Functions
- `tf.diag(diagonal, name=None)`
- `tf.transpose(a, perm=None, name=transpose)`
- `tf.matmul(a, b, transpose_a=False, transpose_b=False, a_is_sparse=False, b_is_sparse=False, name=None)`
- `tf.batch_matmul(x, y, adj_x=None, adj_y=None, name=None)`
- `tf.matrix_determinant(input, name=None)`
- `tf.batch_matrix_determinant(input, name=None)`
- `tf.matrix_inverse(input, name=None)`
- `tf.batch_matrix_inverse(input, name=None)`
- `tf.cholesky(input, name=None)`
- `tf.batch_cholesky(input, name=None)`

- `tf.self_adjoint_eig(input, name=None)`
- `tf.batch_self_adjoint_eig(input, name=None)`
- `tf.matrix_solve(matrix, rhs, name=None)`
- `tf.batch_matrix_solve(matrix, rhs, name=None)`
- `tf.matrix_triangular_solve(matrix, rhs, lower=None, name=None)`
- `tf.batch_matrix_triangular_solve(matrix, rhs, lower=None, name=None)`
- `tf.matrix_solve_ls(matrix, rhs, l2_regularizer=0.0, fast=True, name=None)`
- `tf.batch_matrix_solve_ls(matrix, rhs, l2_regularizer=0.0, fast=True, name=None)`
- Complex Number Functions
- `tf.complex(real, imag, name=None)`
- `tf.complex_abs(x, name=None)`
- `tf.conj(in_, name=None)`
- `tf.imag(in_, name=None)`
- `tf.real(in_, name=None)`
- `tf.fft2d(in_, name=None)`
- `tf.ifft2d(in_, name=None)`
- Reduction
- `tf.reduce_sum(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_prod(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_min(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_max(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_mean(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_all(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.reduce_any(input_tensor, reduction_indices=None, keep_dims=False, name=None)`
- `tf.accumulate_n(inputs, shape=None, tensor_dtype=None, name=None)`
- Segmentation
- `tf.segment_sum(data, segment_ids, name=None)`
- `tf.segment_prod(data, segment_ids, name=None)`
- `tf.segment_min(data, segment_ids, name=None)`
- `tf.segment_max(data, segment_ids, name=None)`
- `tf.segment_mean(data, segment_ids, name=None)`

* `tf.unsorted_segment_sum(data, segment_ids, num_segments, name=None)`
* `tf.sparse_segment_sum(data, indices, segment_ids, name=None)`
* `tf.sparse_segment_mean(data, indices, segment_ids, name=None)`
* `tf.sparse_segment_sqrt_n(data, indices, segment_ids, name=None)`
* Sequence Comparison and Indexing
* `tf.argmin(input, dimension, name=None)`
* `tf.argmax(input, dimension, name=None)`
* `tf.listdiff(x, y, name=None)`
* `tf.where(input, name=None)`
* `tf.unique(x, name=None)`
* `tf.edit_distance(hypothesis, truth, normalize=True, name=edit_distance)`
* `tf.invert_permutation(x, name=None)`
* Other Functions and Classes
* `tf.scalar_mul(scalar, x)`
* `tf.sparse_segment_sqrt_n_grad(grad, indices, segment_ids, output_dim0, name=None)`

Arithmetic Operators

TensorFlow provides several operations that you can use to add basic arithmetic operators to your graph.

`tf.add(x, y, name=None)`

Returns x + y element-wise.

NOTE: Add supports broadcasting. AddN does not.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`,`complex64`, `string`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

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

```
tf.sub(x, y, name=None)
```

Returns x - y element-wise.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

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

`tf.mul(x, y, name=None)`

Returns x * y element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`,`complex64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.div(x, y, name=None)`

Returns x / y element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`,`complex64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.truediv(x, y, name=None)`

Divides x / y elementwise, always producing floating point results.

The same as `tf.div` for floating point arguments, but casts integer arguments to floating point before dividing so that the result is always floating point. This op is generated by normal `x / y` division in Python 3 and in Python 2.7 with `from __future__ import division`. If you want integer division that rounds down, use `x // y`or `tf.floordiv`.

`x` and `y` must have the same numeric type. If the inputs are floating point, the output will have the same type. If the inputs are integral, the inputs are cast to `float32` for `int8` and `int16` and `float64` for `int32` and `int64`(matching the behavior of Numpy).

Args:

- `x`: `Tensor` numerator of numeric type.
- `y`: `Tensor` denominator of numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` evaluated in floating point.

Raises:

- `TypeError`: If `x` and `y` have different dtypes.

`tf.floordiv(x, y, name=None)`

Divides `x / y` elementwise, rounding down for floating point.

The same as `tf.div(x,y)` for integers, but

uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).
`x` and `y` must have the same type, and the result will have the same type as well.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

Raises:

- `TypeError`: If the inputs are complex.

`tf.mod(x, y, name=None)`

Returns element-wise remainder of division.

Args:

- `x`: A `Tensor`. Must be one of the following types: `int32`, `int64`, `float32`, `float64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.cross(a, b, name=None)`

Compute the pairwise cross product.

`a` and `b` must be the same shape; they can either be simple 3-element vectors, or any shape where the innermost dimension is 3. In the latter case, each pair of corresponding 3-element vectors is cross-multiplied independently.

Args:

- `a`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`,`uint16`. A tensor containing 3-element vectors.
- `b`: A `Tensor`. Must have the same type as `a`. Another tensor, of same type and shape as `a`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `a`. Pairwise cross product of the vectors in `a` and `b`.

Basic Math Functions

TensorFlow provides several operations that you can use to add basic mathematical functions to your graph.

```
tf.add_n(inputs, name=None)
```

Add all input tensors element wise.

Args:

- `inputs`: A list of at least 1 `Tensor` objects of the same type in: `float32`, `float64`, `int64`, `int32`, `uint8`,`uint16`, `int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`. Must all be the same size and shape.
- `name`: A name for the operation (optional).

Returns:

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

`tf.abs(x, name=None)`

Computes the absolute value of a tensor.

Given a tensor of real numbers `x`, this operation returns a tensor containing the absolute value of each element in`x`. For example, if x is an input element and y is an output element, this operation computes $y=|x|y=|x|$.

See `tf.complex_abs()` to compute the absolute value of a complex number.

Args:

- `x`: A `Tensor` of type `float`, `double`, `int32`, or `int64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` the same size and type as `x` with absolute values.

`tf.neg(x, name=None)`

Computes numerical negative value element-wise.

I.e., $y = -x$y=−x.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.sign(x, name=None)`

Returns an element-wise indication of the sign of a number.

y = sign(x) = -1 if x < 0; 0 if x == 0; 1 if x > 0.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.inv(x, name=None)`

Computes the reciprocal of x element-wise.

I.e., $y=1/x$y=1/x.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.square(x, name=None)`

Computes square of x element-wise.

I.e., $y = x * x = x^2$.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.round(x, name=None)`

Rounds the values of a tensor to the nearest integer, element-wise.

For example:

```
# 'a' is [0.9, 2.5, 2.3, -4.4]
tf.round(a) ==> [ 1.0, 3.0, 2.0, -4.0 ]
```

Args:

- `x`: A `Tensor` of type `float` or `double`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of same shape and type as `x`.

`tf.sqrt(x, name=None)`

Computes square root of x element-wise.

I.e., $y=\sqrt{x}=x^{1/2}$y=x=x1/2.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.rsqrt(x, name=None)`

Computes reciprocal of square root of x element-wise.

I.e., $y=1/\sqrt{x}$y=1/x.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.pow(x, y, name=None)`

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes x^y for corresponding elements in `x` and `y`. For example:

```
# tensor 'x' is [[2, 2]], [3, 3]]
# tensor 'y' is [[8, 16], [2, 3]]
tf.pow(x, y) ==> [[256, 65536], [9, 27]]
```

Args:

- `x`: A `Tensor` of type `float`, `double`, `int32`, `complex64`, or `int64`.
- `y`: A `Tensor` of type `float`, `double`, `int32`, `complex64`, or `int64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`.

`tf.exp(x, name=None)`

Computes exponential of x element-wise. $y=e^x$y=ex.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.log(x, name=None)`

Computes natural logarithm of x element-wise.

I.e., $y=\log_e x$y=loge x.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.ceil(x, name=None)`

Returns element-wise smallest integer in not less than x.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.floor(x, name=None)`

Returns element-wise largest integer not greater than x.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.maximum(x, y, name=None)`

Returns the max of x and y (i.e. x > y ? x : y) element-wise, broadcasts.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.minimum(x, y, name=None)`

Returns the min of x and y (i.e. x < y ? x : y) element-wise, broadcasts.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

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

`tf.cos(x, name=None)`

Computes cos of x element-wise.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
* `name`: A name for the operation (optional).

Returns:

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

`tf.sin(x, name=None)`

Computes sin of x element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.lgamma(x, name=None)`

Computes `ln(|gamma(x)|)` element-wise.

Args:

- `x`: A Tensor with type `float`, `double`, `int32`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

`tf.erf(x, name=None)`

Computes Gauss error function of `x` element-wise.

Args:

- `x`: A Tensor with type `float`, `double`, `int32`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

`tf.erfc(x, name=None)`

Computes complementary error function of `x` element-wise.

Args:

- `x`: A Tensor with type `float`, `double`, `int32`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

Matrix Math Functions

TensorFlow provides several operations that you can use to add basic mathematical functions for matrices to your graph.

`tf.diag(diagonal, name=None)`

Returns a diagonal tensor with a given diagonal values.

Given a `diagonal`, this operation returns a tensor with

the `diagonal` and everything else padded with zeros. The diagonal is computed as follows:

Assume `diagonal` has dimensions [D1,..., Dk], then the output is a tensor of rank 2k with dimensions [D1,..., Dk, D1,..., Dk] where:

`output[i1,..., ik, i1,..., ik] = diagonal[i1, ..., ik]` and 0 everywhere else.

For example:

```
# 'diagonal' is [1, 2, 3, 4]
tf.diag(diagonal) ==> [[1, 0, 0, 0]
                       [0, 2, 0, 0]
                       [0, 0, 3, 0]
                       [0, 0, 0, 4]]
```

Args:

- `diagonal`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`. Rank k tensor where k is at most 3.
- `name`: A name for the operation (optional).

Returns:

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

`tf.transpose(a, perm=None, name='transpose')`

Transposes `a`. Permutes the dimensions according to `perm`.

The returned tensor's dimension i will correspond to the input dimension `perm[i]`. If `perm` is not given, it is set to (n-1...0), where n is the rank of the input tensor. Hence by default, this operation performs a regular matrix transpose on 2-D input Tensors.

For example:

```
# 'x' is [[1 2 3]
#         [4 5 6]]
tf.transpose(x) ==> [[1 4]
                     [2 5]
                     [3 6]]

# Equivalently
tf.transpose(x, perm=[1, 0]) ==> [[1 4]
                                  [2 5]
                                  [3 6]]

# 'perm' is more useful for n-dimensional tensors, for n > 2
# 'x' is   [[[1  2  3]
#            [4  5  6]]
#           [[7  8  9]
#            [10 11 12]]]
# Take the transpose of the matrices in dimension-0
tf.transpose(b, perm=[0, 2, 1]) ==> [[[1  4]
                                      [2  5]
                                      [3  6]]

                                     [[7 10]
                                      [8 11]
                                      [9 12]]]
```

Args:

- `a`: A `Tensor`.
- `perm`: A permutation of the dimensions of `a`.
- `name`: A name for the operation (optional).

Returns:

A transposed `Tensor`.

`tf.matmul(a, b, transpose_a=False, transpose_b=False, a_is_sparse=False, b_is_sparse=False, name=None)`

Multiplies matrix `a` by matrix `b`, producing `a` * `b`.

The inputs must be two-dimensional matrices, with matching inner dimensions, possibly after transposition.

Both matrices must be of the same type. The supported types are: `float`, `double`, `int32`, `complex64`.

Either matrix can be transposed on the fly by setting the corresponding flag to `True`. This is `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default.

For example:

```
# 2-D tensor `a`
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]) => [[1. 2. 3.]
                                                      [4. 5. 6.]]
```

```
# 2-D tensor `b`
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2]) => [[7. 8.]
                                                          [9. 10.]
                                                          [11. 12.]]
c = tf.matmul(a, b) => [[58 64]
                        [139 154]]
```

Args:

- `a`: `Tensor` of type `float`, `double`, `int32` or `complex64`.
- `b`: `Tensor` with same type as `a`.
- `transpose_a`: If `True`, `a` is transposed before multiplication.
- `transpose_b`: If `True`, `b` is transposed before multiplication.
- `a_is_sparse`: If `True`, `a` is treated as a sparse matrix.
- `b_is_sparse`: If `True`, `b` is treated as a sparse matrix.
- `name`: Name for the operation (optional).

Returns:

A `Tensor` of the same type as `a`.

`tf.batch_matmul(x, y, adj_x=None, adj_y=None, name=None)`

Multiplies slices of two tensors in batches.

Multiplies all slices of `Tensor` `x` and `y` (each slice can be viewed as an element of a batch), and arranges the individual results in a single output tensor of the same batch size. Each of the individual slices can optionally be adjointed (to adjoint a matrix means to transpose and conjugate it) before multiplication by setting the `adj_x` or`adj_y` flag to `True`, which are by default `False`.

The input tensors `x` and `y` are 3-D or higher with shape `[..., r_x, c_x]` and `[..., r_y, c_y]`.

The output tensor is 3-D or higher with shape `[..., r_o, c_o]`, where:

```
r_o = c_x if adj_x else r_x
c_o = r_y if adj_y else c_y
```

It is computed as:

```
out[..., :, :] = matrix(x[..., :, :]) * matrix(y[..., :, :])
```

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `complex64`. 3-D or higher with shape `[..., r_x, c_x]`.
- `y`: A `Tensor`. Must have the same type as `x`. 3-D or higher with shape `[..., r_y, c_y]`.
- `adj_x`: An optional `bool`. Defaults to `False`. If `True`, adjoint the slices of `x`. Defaults to `False`.
- `adj_y`: An optional `bool`. Defaults to `False`. If `True`, adjoint the slices of `y`. Defaults to `False`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`. 3-D or higher with shape `[..., r_o, c_o]`

`tf.matrix_determinant(input, name=None)`

Calculates the determinant of a square matrix.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`. A tensor of shape `[M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. A scalar, equal to the determinant of the input.

`tf.batch_matrix_determinant(input, name=None)`

Calculates the determinants for a batch of square matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. The output is a 1-D tensor containing the determinants for all input submatrices `[..., :, :]`.

Args:

- `input`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`. Shape is `[..., M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[...]`.

`tf.matrix_inverse(input, name=None)`

Calculates the inverse of a square invertible matrix.

The op uses the Cholesky decomposition if the matrix is symmetric positive definite and LU decomposition with partial pivoting otherwise.

If the matrix is not invertible there is no guarantee what the op does. It may detect the condition and raise an exception or it may simply return a garbage result.

Args:

- `input`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`. Shape is `[M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[M, M]` containing the matrix inverse of the input.

`tf.batch_matrix_inverse(input, name=None)`

Calculates the inverse of square invertible matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the inverse for all input submatrices `[..., :, :]`.

The op uses the Cholesky decomposition if the matrices are symmetric positive definite and LU decomposition with partial pivoting otherwise.

If a matrix is not invertible there is no guarantee what the op does. It may detect the condition and raise an exception or it may simply return a garbage result.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[..., M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[..., M, M]`.

`tf.cholesky(input, name=None)`

Calculates the Cholesky decomposition of a square matrix.

The input has to be symmetric and positive definite. Only the lower-triangular part of the input will be used for this operation. The upper-triangular part will not be read.

The result is the lower-triangular matrix of the Cholesky decomposition of the input.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[M, M]`.

`tf.batch_cholesky(input, name=None)`

Calculates the Cholesky decomposition of a batch of square matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices, with the same constraints as the single matrix Cholesky decomposition above. The output is a tensor of the same shape as the input containing the Cholesky decompositions for all input submatrices `[..., :, :]`.

Args:

- `input`: A `Tensor`. Must be one of the following

  types: `float64`, `float32`. Shape is `[..., M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[..., M, M]`.

`tf.self_adjoint_eig(input, name=None)`

Calculates the Eigen Decomposition of a square Self-Adjoint matrix.

Only the lower-triangular part of the input will be used in this case. The upper-triangular part will not be read.

The result is a M+1 x M matrix whose first row is the eigenvalues, and subsequent rows are eigenvectors.

Args:

- `input`: A `Tensor`. Must be one of the following

  types: `float64`, `float32`. Shape is `[M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[M+1, M]`.

```
tf.batch_self_adjoint_eig(input, name=None)
```

Calculates the Eigen Decomposition of a batch of square self-adjoint matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices, with the same constraints as the single matrix SelfAdjointEig.

The result is a '[..., M+1, M] matrix with [..., 0,:] containing the eigenvalues, and subsequent [...,1:, :] containing the eigenvectors.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[..., M, M]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Shape is `[..., M+1, M]`.

```
tf.matrix_solve(matrix, rhs, name=None)
```

Solves a system of linear equations. Checks for invertibility.

Args:

- `matrix`: A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[M, M]`.

- `rhs`: A `Tensor`. Must have the same type as `matrix`. Shape is `[M, K]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[M, K]` containing the tensor that solves matrix * output = rhs.

`tf.batch_matrix_solve(matrix, rhs, name=None)`

Solves systems of linear equations. Checks for invertibility.

Matrix is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. Rhs is a tensor of shape `[..., M, K]`. The output is a tensor shape `[..., M, K]` where each output matrix satisfies matrix[..., :, :] * output[..., :, :] = rhs[..., :, :].

Args:

- `matrix`: A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[..., M, M]`.
- `rhs`: A `Tensor`. Must have the same type as `matrix`. Shape is `[..., M, K]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[..., M, K]`.

`tf.matrix_triangular_solve(matrix, rhs, lower=None, name=None)`

Solves a system of linear equations with an upper or lower triangular matrix by

backsubstitution.

`matrix` is a matrix of shape `[M, M]`. If `lower` is `True` then the strictly upper triangular part of `matrix` is ignored. If `lower` is False then the strictly lower triangular part of `matrix` is ignored. `rhs` is a matrix of shape [M, K]`.

The output is a matrix of shape `[M, K]`. If `lower` is `True` then the output satisfies $\sum_{k=0}^{i}$ matrix[i, k] * output[k, j] = rhs[i, j]. If `lower` is false then output satisfies $\sum_{k=i}^{K-1}$ matrix[i, k] * output[k, j] = rhs[i, j].

Args:

- `matrix`: A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[M, M]`.
- `rhs`: A `Tensor`. Must have the same type as `matrix`. Shape is `[M, K]`.

* `lower`: An optional `bool`. Defaults to `True`. Boolean indicating whether matrix is lower or upper triangular.
* `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[M, K]`.

`tf.batch_matrix_triangular_solve(matrix, rhs, lower=None, name=None)`

Solves systems of linear equations with upper or lower triangular matrices by

backsubstitution.

`matrix` is a tensor of shape `[..., M, M]` whose inner-most 2

dimensions form square matrices. If `lower` is`True` then the strictly

upper triangular part of each inner-most matrix is ignored. If `lower` is False then the strictly lower triangular part of each inner-most matrix is ignored. `rhs` is a tensor of shape [..., M, K]`.

The output is a tensor of shape `[..., M, K]`. If `lower` is `True` then

the output satisfies $\sum_{k=0}^{i}$∑k=0i matrix[..., i, k] * output[..., k, j] = rhs[...,

i, j]. If `lower` is false then the strictly then the output

satisfies $sum_{k=i}^{K-1}$sumk=iK−1 matrix[..., i, k] * output[..., k, j] = rhs[..., i, j].

Args:

- `matrix`: A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[..., M, M]`.
- `rhs`: A `Tensor`. Must have the same type as `matrix`. Shape is `[..., M, K]`.
- `lower`: An optional `bool`. Defaults to `True`. Boolean indicating whether matrix is lower or upper triangular.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[..., M, K]`.

```
tf.matrix_solve_ls(matrix, rhs, l2_regularizer=0.0,
fast=True, name=None)
```

Solves a linear least-squares problem.

Below we will use the following notation `matrix`$=A \in \mathbb{R}^{m \times n} A \in \Re^{m \times n}$, `rhs`$=B \in \mathbb{R}^{m \times k} B \in \Re^{m \times k}$, `output`$=X \in \mathbb{R}^{n \times k} X \in \Re^{n \times k}$,`l2_regularizer`$=\lambda\lambda$.

If `fast` is `True`, then the solution is computed by solving the normal equations using Cholesky decomposition. Specifically, if $m \geq n m \geq n$ then $X = (A^T A + \lambda I)^{-1} A^T B X = (A^T A + \lambda I)^{-1} A^T B$, which solves the regularized least-squares problem $X = \mathrm{argmin}_{Z \in \mathbb{R}^{n \times k}} ||AZ - B||_F^2 + \lambda ||Z||_F^2 X = \mathrm{argmin}_{Z \in \Re^{n \times k}} ||AZ - B||_F^2 + \lambda ||Z||_F^2$.

If $m<n$ then `output` is computed

as $X=A^T(AA^T+\lambda I)^{-1}B$, which (for $\lambda=0$) is the minimum-norm solution to the under-determined linear system, i.e. $X=\mathrm{argmin}_{Z\in\Re^{n\times k}}||Z||_F^2$, subject

to $AZ=B$. Notice that the fast path is only numerically stable

when A is numerically full rank and has a condition

number $\mathrm{cond}(A)<\frac{1}{\sqrt{\epsilon_{mach}}}$ or λ is sufficiently large.

If `fast` is `False` then the solution is computed using the rank revealing QR decomposition with column pivoting. This will always compute a least-squares solution that minimizes the residual norm $||AX-B||_F^2$, even when $A$is rank deficient or ill-conditioned. Notice: The current version does not compute a minimum norm solution. If`fast` is `False` then `l2_regularizer` is ignored.

Args:

- `matrix`: 2-D `Tensor` of shape `[M, N]`.
- `rhs`: 2-D `Tensor` of shape is `[M, K]`.
- `l2_regularizer`: 0-D `double` `Tensor`. Ignored if `fast=False`.
- `fast`: bool. Defaults to `True`.
- `name`: string, optional name of the operation.

Returns:

- `output`: Matrix of shape `[N, K]` containing the matrix that solves `matrix * output = rhs` in the least-squares sense.

```
tf.batch_matrix_solve_ls(matrix, rhs, l2_regularizer=0.0, fast=True, name=None)
```

Solves multiple linear least-squares problems.

`matrix` is a tensor of shape `[..., M, N]` whose inner-most 2 dimensions form `M`-by-`N` matrices. Rhs is a tensor of shape `[..., M, K]` whose inner-most 2 dimensions form `M`-by-`K` matrices. The computed output is a`Tensor` of shape `[..., N, K]` whose inner-most 2 dimensions form `M`-by-`K` matrices that solve the equations`matrix[..., :, :] * output[..., :, :] = rhs[..., :, :]` in the least squares sense.

Below we will use the following notation for each pair of matrix and right-hand sides in the batch:

`matrix`$=A \in \Re^{m \times n}$, `rhs`$=B \in \Re^{m \times k}$, `output`$=X \in \Re^{n \times k}$, `l2_regularizer`$=\lambda$.

If `fast` is `True`, then the solution is computed by solving the normal equations using Cholesky decomposition. Specifically, if $m \ge n$ then $X = (A^T A + \lambda I)^{-1} A^T B$, which solves the least-squares problem $X = \mathrm{argmin}_{Z \in \Re^{n \times k}} ||A Z - B||_F^2 + \lambda ||Z||_F^2$. If $m < n$ then `output` is computed as $X = A^T (A A^T + \lambda I)^{-1} B$, which (for $\lambda = 0$) is the minimum-norm solution to the under-determined linear system, i.e. $X = \mathrm{argmin}_{Z \in \Re^{n \times k}} ||Z||_F^2$, subject to $AZ=B$. Notice that the fast path is only numerically stable when A is numerically full rank and has a condition number $\mathrm{cond}(A) < \frac{1}{\sqrt{\epsilon_{mach}}}$ or λ is sufficiently large.

If `fast` is `False` then the solution is computed using the rank revealing QR decomposition with column pivoting. This will always compute a least-squares solution that minimizes the residual norm $||AX-B||_F^2$, even when A is rank deficient or ill-conditioned. Notice: The current version does not compute a minimum norm solution. If `fast` is `False` then `l2_regularizer` is ignored.

Args:

- `matrix`: `Tensor` of shape `[..., M, N]`.
- `rhs`: `Tensor` of shape `[..., M, K]`.
- `l2_regularizer`: 0-D `double` `Tensor`. Ignored if `fast=False`.
- `fast`: bool. Defaults to `True`.
- `name`: string, optional name of the operation.

Returns:

- `output`: `Tensor` of shape `[..., N, K]` whose inner-most 2 dimensions form `M`-by-`K` matrices that solve the equations `matrix[..., :, :] * output[..., :, :] = rhs[..., :, :]` in the least squares sense.

Complex Number Functions

TensorFlow provides several operations that you can use to add complex number functions to your graph.

`tf.complex(real, imag, name=None)`

Converts two real numbers to a complex number.

Given a tensor `real` representing the real part of a complex number, and a tensor `imag` representing the imaginary part of a complex number, this operation computes complex numbers elementwise of the form a+bja+bj, where a represents the `real` part and b represents the `imag` part.

The input tensors `real` and `imag` must be the same shape.

For example:

```
# tensor 'real' is [2.25, 3.25]
# tensor `imag` is [4.75, 5.75]
tf.complex(real, imag) ==> [[2.25 + 4.74j], [3.25 + 5.75j]]
```

Args:

- `real`: A `Tensor` of type `float`.
- `imag`: A `Tensor` of type `float`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`.

`tf.complex_abs(x, name=None)`

Computes the complex absolute value of a tensor.

Given a tensor `x` of complex numbers, this operation returns a tensor of type `float` that is the absolute value of each element in `x`. All elements in `x` must be complex numbers of the form $a + bj$. The absolute value is computed as $\sqrt{a^2 + b^2}$.

For example:

```
# tensor 'x' is [[-2.25 + 4.75j], [-3.25 + 5.75j]]
tf.complex_abs(x) ==> [5.25594902, 6.60492229]
```

Args:

- `x`: A `Tensor` of type `complex64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

`tf.conj(in_, name=None)`

Returns the complex conjugate of a complex number.

Given a tensor `in` of complex numbers, this operation returns a tensor of complex numbers that are the complex conjugate of each element in `in`. The complex numbers in `in` must be of the

form $a+bj$a+bj, where a is the real part and b is the imaginary part. The complex conjugate returned by this operation is of the form $a-bj$a−bj.

For example:

```
# tensor 'in' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.conj(in) ==> [-2.25 - 4.75j, 3.25 - 5.75j]
```

Args:

- `in_`: A `Tensor` of type `complex64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`.

`tf.imag(in_, name=None)`

Returns the imaginary part of a complex number.

Given a tensor `in` of complex numbers, this operation returns a tensor of type `float` that is the imaginary part of each element in `in`.

All elements in `in` must be complex numbers of the form a+bja+bj, where a is the real part and bis the imaginary part returned by this operation.

For example:

```
# tensor 'in' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.imag(in) ==> [4.75, 5.75]
```

Args:

- `in_`: A `Tensor` of type `complex64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

`tf.real(in_, name=None)`

Returns the real part of a complex number.

Given a tensor `in` of complex numbers, this operation returns a tensor of type `float` that is the real part of each element in `in`. All elements in `in` must be complex numbers of the form a+bja+bj, where a is the real part returned by this operation and b is the imaginary part.

For example:

```
# tensor 'in' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.real(in) ==> [-2.25, 3.25]
```

Args:

- `in_`: A `Tensor` of type `complex64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

`tf.fft2d(in_, name=None)`

Compute the 2-dimensional discrete Fourier Transform.

Args:

- `in_`: A `Tensor` of type `complex64`. A complex64 matrix.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`. The 2D Fourier Transform of `in`.

`tf.ifft2d(in_, name=None)`

Compute the inverse 2-dimensional discrete Fourier Transform.

Args:

- `in_`: A `Tensor` of type `complex64`. A complex64 matrix.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`. The inverse 2D Fourier Transform of `in`.

Reduction

TensorFlow provides several operations that you can use to perform common math computations that reduce various dimensions of a tensor.

`tf.reduce_sum(input_tensor, reduction_indices=None, keep_dims=False, name=None)`

Computes the sum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given

in `reduction_indices`. Unless `keep_dims` is true, the rank of the

tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
# 'x' is [[1, 1, 1]
#         [1, 1, 1]]
```

```
tf.reduce_sum(x) ==> 6
tf.reduce_sum(x, 0) ==> [2, 2, 2]
tf.reduce_sum(x, 1) ==> [3, 3]
tf.reduce_sum(x, 1, keep_dims=True) ==> [[3], [3]]
tf.reduce_sum(x, [0, 1]) ==> 6
```

Args:

- `input_tensor`: The tensor to reduce. Should have numeric type.
- `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:

The reduced tensor.

`tf.reduce_prod(input_tensor, reduction_indices=None, keep_dims=False, name=None)`

Computes the product of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given

in `reduction_indices`. Unless `keep_dims` is true, the rank of the

tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

Args:

- `input_tensor`: The tensor to reduce. Should have numeric type.
- `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:

The reduced tensor.

```
tf.reduce_min(input_tensor, reduction_indices=None,
keep_dims=False, name=None)
```

Computes the minimum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.
If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

Args:

- `input_tensor`: The tensor to reduce. Should have numeric type.

* `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
* `keep_dims`: If true, retains reduced dimensions with length 1.
* `name`: A name for the operation (optional).

Returns:

The reduced tensor.

```
tf.reduce_max(input_tensor, reduction_indices=None,
keep_dims=False, name=None)
```

Computes the maximum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

Args:

* `input_tensor`: The tensor to reduce. Should have numeric type.
* `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
* `keep_dims`: If true, retains reduced dimensions with length 1.

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

Returns:

The reduced tensor.

`tf.reduce_mean(input_tensor, reduction_indices=None, keep_dims=False, name=None)`

Computes the mean of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.
If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
# 'x' is [[1., 1.]
#         [2., 2.]]
tf.reduce_mean(x) ==> 1.5
tf.reduce_mean(x, 0) ==> [1.5, 1.5]
tf.reduce_mean(x, 1) ==> [1.,  2.]
```

Args:

- `input_tensor`: The tensor to reduce. Should have numeric type.

- `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:

The reduced tensor.

```
tf.reduce_all(input_tensor, reduction_indices=None,
keep_dims=False, name=None)
```

Computes the "logical and" of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
# 'x' is [[True,  True]
#         [False, False]]
tf.reduce_all(x) ==> False
tf.reduce_all(x, 0) ==> [False, False]
tf.reduce_all(x, 1) ==> [True, False]
```

Args:

- `input_tensor`: The boolean tensor to reduce.
- `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:

The reduced tensor.

`tf.reduce_any(input_tensor, reduction_indices=None, keep_dims=False, name=None)`

Computes the "logical or" of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given

in `reduction_indices`. Unless `keep_dims` is true, the rank of the

tensor is reduced by 1 for each entry in `reduction_indices`.

If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
# 'x' is [[True,  True]
#         [False, False]]
tf.reduce_any(x) ==> True
tf.reduce_any(x, 0) ==> [True, True]
```

```
tf.reduce_any(x, 1) ==> [True, False]
```

Args:

- `input_tensor`: The boolean tensor to reduce.
- `reduction_indices`: The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:

The reduced tensor.

```
tf.accumulate_n(inputs, shape=None, tensor_dtype=None,
name=None)
```

Returns the element-wise sum of a list of tensors.

Optionally, pass `shape` and `tensor_dtype` for shape and type checking, otherwise, these are inferred.

For example:

```
# tensor 'a' is [[1, 2], [3, 4]
# tensor `b` is [[5, 0], [0, 6]]
tf.accumulate_n([a, b, a]) ==> [[7, 4], [6, 14]]

# Explicitly pass shape and type
tf.accumulate_n([a, b, a], shape=[2, 2], tensor_dtype=tf.int32)
  ==> [[7, 4], [6, 14]]
```

Args:

- `inputs`: A list of `Tensor` objects, each with same shape and type.
- `shape`: Shape of elements of `inputs`.
- `tensor_dtype`: The type of `inputs`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of same shape and type as the elements of `inputs`.

Raises:

- `ValueError`: If `inputs` don't all have same shape and dtype or the shape cannot be inferred.

Segmentation

TensorFlow provides several operations that you can use to perform common math computations on tensor segments. Here a segmentation is a partitioning of a tensor along the first dimension, i.e. it defines a mapping from the first dimension onto `segment_ids`.

The `segment_ids` tensor should be the size of the first dimension,`d0`, with consecutive IDs in the range `0` to `k`, where `k<d0`. In particular, a segmentation of a matrix tensor is a mapping of rows to segments.

For example:

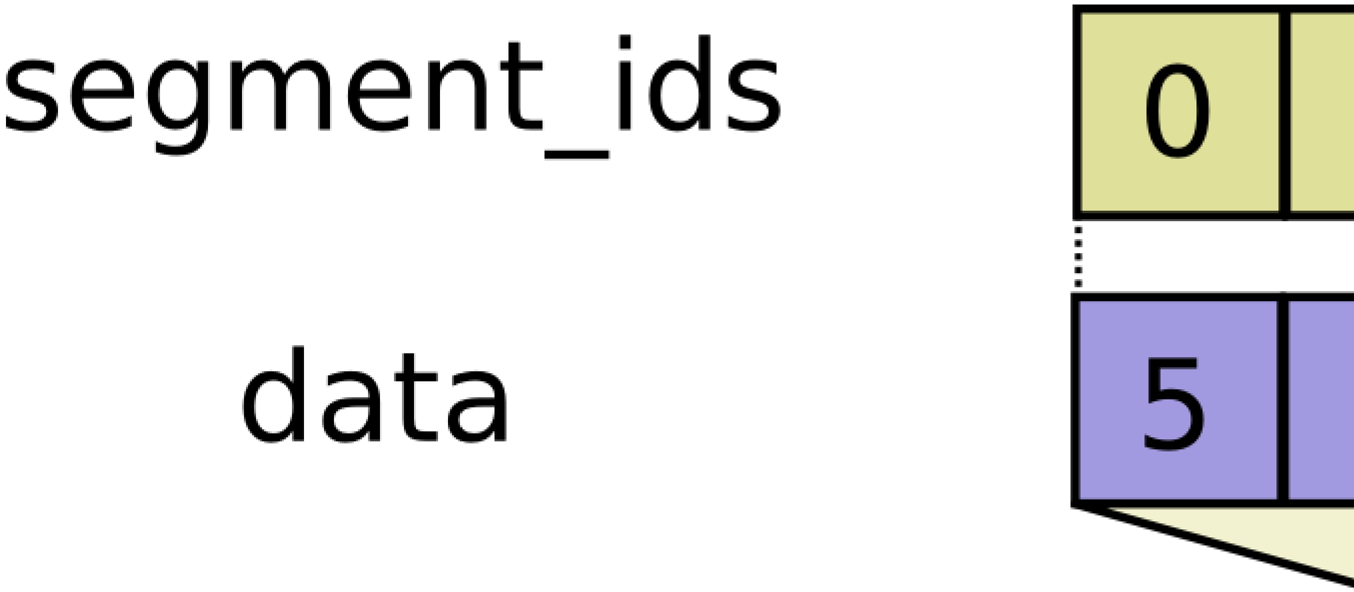
```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])
tf.segment_sum(c, tf.constant([0, 0, 1]))
  ==>  [[0 0 0 0]
        [5 6 7 8]]
```

`tf.segment_sum(data, segment_ids, name=None)`

Computes the sum along segments of a tensor.

Read the section on Segmentation for an explanation of segments.

Computes a tensor such that $output_i = \sum_j data_j$ where sum is over `j` such that `segment_ids[j] == i`.



Args:

- `data`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `segment_ids`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank

of `data`'s first dimension. Values should be sorted and can be repeated.

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

Returns:

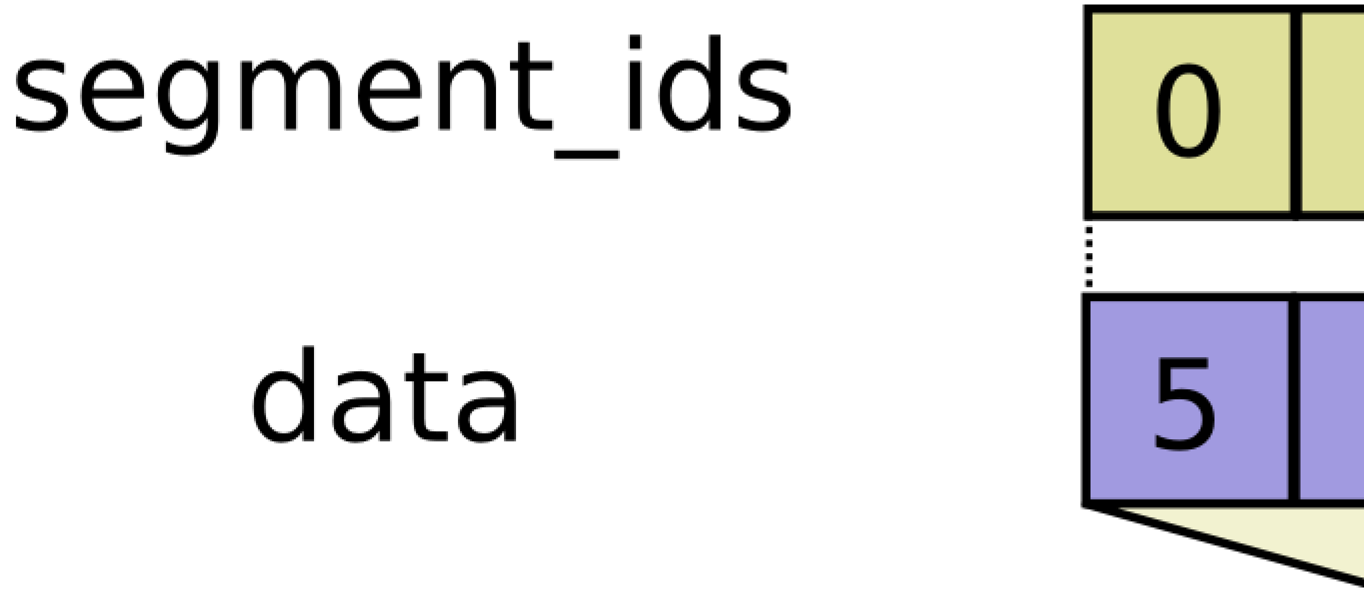
A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

`tf.segment_prod(data, segment_ids, name=None)`

Computes the product along segments of a tensor.

Read the section on Segmentation for an explanation of segments.

Computes a tensor such that $output_i = \prod_j data_j$ where the product is over `j` such that `segment_ids[j] == i`.



Args:

- `data`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `segment_ids`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

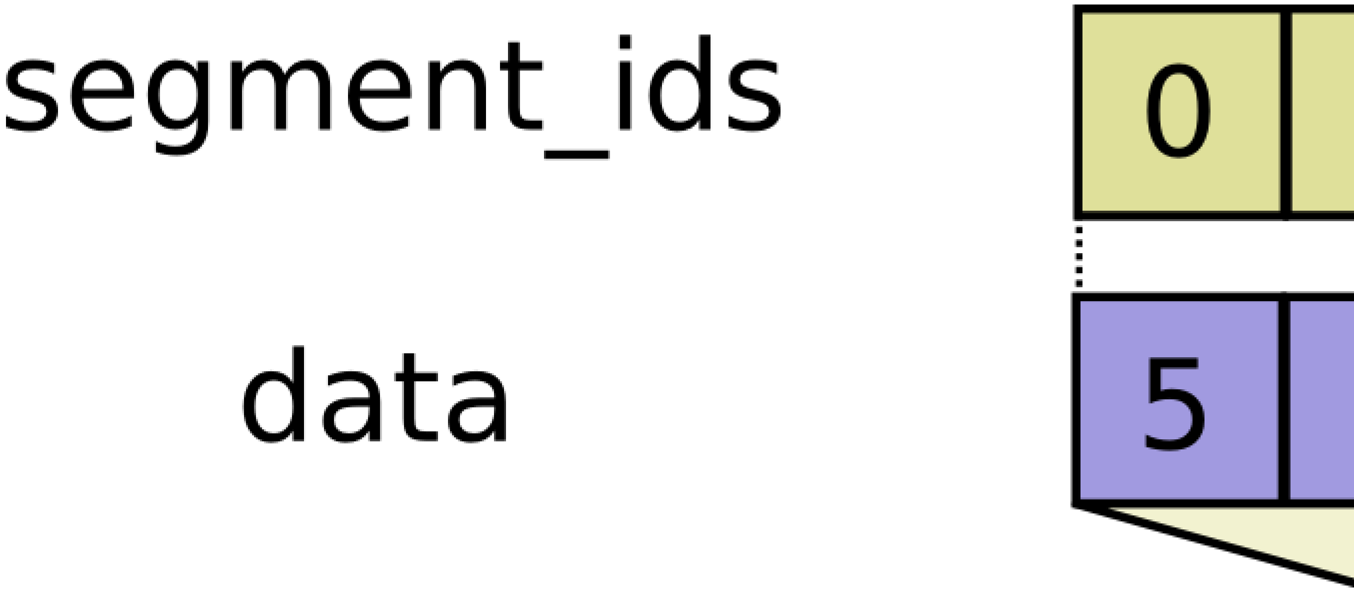
`tf.segment_min(data, segment_ids, name=None)`

Computes the minimum along segments of a tensor.

Read [the section on Segmentation]() for an explanation of segments. Computes a tensor such

that $output_i = \min_j(data_j)$ where `min` is over `j` such

that `segment_ids[j] == i`.



Args:

- `data`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `segment_ids`: A `Tensor`. Must be one of the following

  types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.

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

Returns:

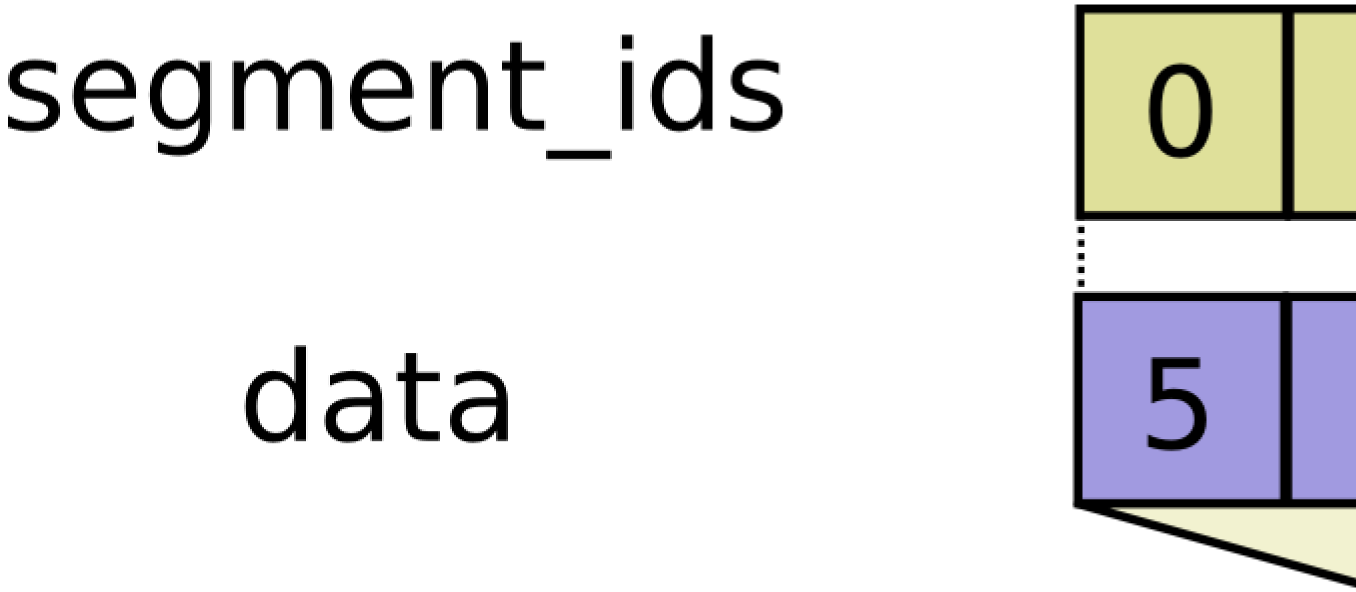
A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

`tf.segment_max(data, segment_ids, name=None)`

Computes the maximum along segments of a tensor.

Read the section on Segmentation for an explanation of segments. Computes a tensor such

that $output_i = \max_j(data_j)$output_i=max_j(data_j) where `max` is over `j` such that `segment_ids[j] == i`.



Args:

- `data`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `segment_ids`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

`tf.segment_mean(data, segment_ids, name=None)`

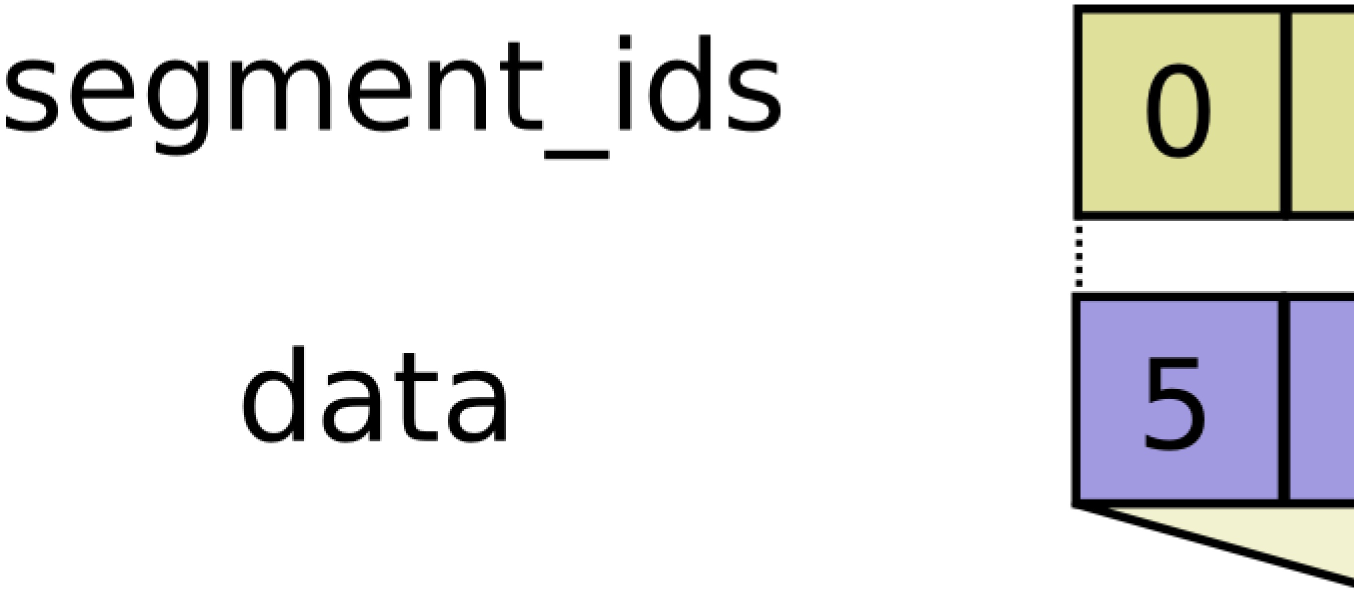
Computes the mean along segments of a tensor.

Read [the section on Segmentation]() for an explanation of segments.
Computes a tensor such

that $output_i=\frac{\sum_j data_j}{N}$ $output_i=\frac{\sum_j data_j}{N}$ where `mean` is over `j` such

that `segment_ids[j] == i` and `N` is the total number of values

summed.



Args:

- `data`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.

- `segment_ids`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

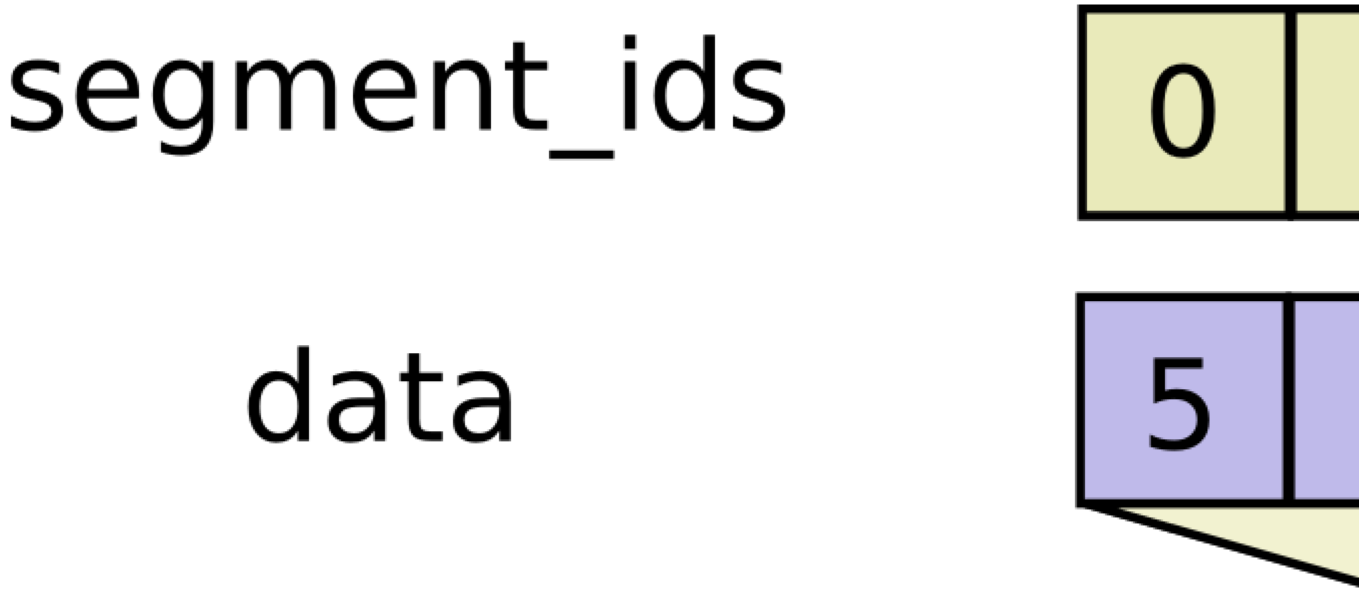
`tf.unsorted_segment_sum(data, segment_ids, num_segments, name=None)`

Computes the sum along segments of a tensor.

Read the section on Segmentation for an explanation of segments.
Computes a tensor such that $output_i=\sum_j data_j$ $output_i=\sum_j data_j$ where sum is over `j` such that `segment_ids[j] == i`.

Unlike `SegmentSum`, `segment_ids` need not be sorted and need not cover all values in the full range of valid values.
If the sum is empty for a given segment ID `i`, `output[i] = 0`.

`num_segments` should equal the number of distinct segment IDs.



Args:

- `data`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `segment_ids`: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension.
- `num_segments`: A `Tensor` of type `int32`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size`num_segments`.

```
tf.sparse_segment_sum(data, indices, segment_ids,
name=None)
```

Computes the sum along sparse segments of a tensor.

Read the section on Segmentation for an explanation of segments. Like `SegmentSum`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`.

For example:

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])

# Select two rows, one segment.
tf.sparse_segment_sum(c, tf.constant([0, 1]), tf.constant([0,
0]))
  ==> [[0 0 0 0]]

# Select two rows, two segment.
tf.sparse_segment_sum(c, tf.constant([0, 1]), tf.constant([0,
1]))
  ==> [[ 1  2  3  4]
       [-1 -2 -3 -4]]

# Select all rows, two segments.
tf.sparse_segment_sum(c, tf.constant([0, 1, 2]), tf.constant([0,
0, 1]))
  ==> [[0 0 0 0]
       [5 6 7 8]]

# Which is equivalent to:
tf.segment_sum(c, tf.constant([0, 0, 1]))
```

Args:

- `data`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`,`int8`, `uint16`.
- `indices`: A `Tensor` of type `int32`. A 1-D tensor. Has same rank as `segment_ids`.
- `segment_ids`: A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

```
tf.sparse_segment_mean(data, indices, segment_ids,
name=None)
```

Computes the mean along sparse segments of a tensor.

Read the section on Segmentation for an explanation of segments. Like `SegmentMean`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`.

Args:

- `data`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`.
- `indices`: A `Tensor` of type `int32`. A 1-D tensor. Has same rank

  as `segment_ids`.
- `segment_ids`: A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data,

except for dimension 0 which has size `k`, the number of segments.

```
tf.sparse_segment_sqrt_n(data, indices, segment_ids,
name=None)
```

Computes the sum along sparse segments of a tensor divided by the sqrt of N.

N is the size of the segment being reduced.

Read the section on Segmentation for an explanation of segments.

Args:

- `data`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`.

* `indices`: A `Tensor` of type `int32`. A 1-D tensor. Has same rank as `segment_ids`.
* `segment_ids`: A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.
* `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

Sequence Comparison and Indexing

TensorFlow provides several operations that you can use to add sequence comparison and index extraction to your graph. You can use these operations to determine sequence differences and determine the indexes of specific values in a tensor.

`tf.argmin(input, dimension, name=None)`

Returns the index with the smallest value across dimensions of a tensor.

Args:

* `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`,`uint16`, `int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`.

- `dimension`: A `Tensor` of type `int32`. int32, 0 <= dimension <

  rank(input). Describes which dimension of the input Tensor to reduce across. For vectors, use dimension = 0.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

`tf.argmax(input, dimension, name=None)`

Returns the index with the largest value across dimensions of a tensor.

Args:

- `input`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int64`, `int32`, `uint8`,`uint16`, `int16`, `int8`,

  `complex64`, `qint8`, `quint8`, `qint32`.
- `dimension`: A `Tensor` of type `int32`. int32, 0 <= dimension <

  rank(input). Describes which dimension of the input Tensor to reduce across. For vectors, use dimension = 0.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

`tf.listdiff(x, y, name=None)`

Computes the difference between two lists of numbers or strings.

Given a list `x` and a list `y`, this operation returns a list `out` that represents all values that are in `x` but not in `y`. The returned list `out` is sorted in the same order that the numbers appear in `x` (duplicates are preserved). This operation also returns a list `idx` that represents the position of each `out` element in `x`. In other words:
`out[i] = x[idx[i]] for i in [0, 1, ..., len(out) - 1]`

For example, given this input:

```
x = [1, 2, 3, 4, 5, 6]
y = [1, 3, 5]
```

This operation would return:

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

Args:

- `x`: A `Tensor`. 1-D. Values to keep.
- `y`: A `Tensor`. Must have the same type as `x`. 1-D. Values to remove.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (out, idx).

- `out`: A `Tensor`. Has the same type as `x`. 1-D. Values present in `x` but not in `y`.
- `idx`: A `Tensor` of type `int32`. 1-D. Positions of `x` values preserved in `out`.

`tf.where(input, name=None)`

Returns locations of true values in a boolean tensor.

This operation returns the coordinates of true elements in `input`. The coordinates are returned in a 2-D tensor where the first dimension (rows) represents the number of true elements, and the second dimension (columns) represents the coordinates of the true elements. Keep in mind, the shape of the output tensor can vary depending on how many true values there are in `input`. Indices are output in row-major order.

For example:

```
# 'input' tensor is [[True, False]
#                    [True, False]]
# 'input' has two true values, so output has two coordinates.
# 'input' has rank of 2, so coordinates have two indices.
where(input) ==> [[0, 0],
                  [1, 0]]

# `input` tensor is [[[True, False]
#                     [True, False]]
#                    [[False, True]
#                     [False, True]]
#                    [[False, False]
#                     [False, True]]]
# 'input' has 5 true values, so output has 5 coordinates.
# 'input' has rank of 3, so coordinates have three indices.
where(input) ==> [[0, 0, 0],
```

```
              [0, 1, 0],
              [1, 0, 1],
              [1, 1, 1],
              [2, 1, 1]]
```

Args:

- `input`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

`tf.unique(x, name=None)`

Finds unique elements in a 1-D tensor.

This operation returns a tensor `y` containing all of the unique elements of `x` sorted in the same order that they occur in `x`. This operation also returns a tensor `idx` the same size as `x` that contains the index of each value of `x`in the unique output `y`. In other words:

`y[idx[i]] = x[i] for i in [0, 1,...,rank(x) - 1]`

For example:

```
# tensor 'x' is [1, 1, 2, 4, 4, 4, 7, 8, 8]
y, idx = unique(x)
y ==> [1, 2, 4, 7, 8]
idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
```

Args:

- `x`: A `Tensor`. 1-D.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (y, idx).

- `y`: A `Tensor`. Has the same type as `x`. 1-D.
- `idx`: A `Tensor` of type `int32`. 1-D.

`tf.edit_distance(hypothesis, truth, normalize=True, name='edit_distance')`

Computes the Levenshtein distance between sequences.

This operation takes variable-length sequences (`hypothesis` and `truth`), each provided as a `SparseTensor`, and computes the Levenshtein distance. You can normalize the edit distance by length of `truth` by setting`normalize` to true.

For example, given the following input:

```
# 'hypothesis' is a tensor of shape `[2, 1]` with variable-length values:
#   (0,0) = ["a"]
#   (1,0) = ["b"]
hypothesis = tf.SparseTensor(
    [[0, 0, 0],
     [1, 0, 0]],
    ["a", "b"]
    (2, 1, 1))
```

```
# 'truth' is a tensor of shape `[2, 2]` with variable-length
values:
#   (0,0) = []
#   (0,1) = ["a"]
#   (1,0) = ["b", "c"]
#   (1,1) = ["a"]
truth = tf.SparseTensor(
    [[0, 1, 0],
     [1, 0, 0],
     [1, 0, 1],
     [1, 1, 0]]
    ["a", "b", "c", "a"],
    (2, 2, 2))

normalize = True
```

This operation would return the following:

```
# 'output' is a tensor of shape `[2, 2]` with edit distances
normalized
# by 'truth' lengths.
output ==> [[inf, 1.0],  # (0,0): no truth, (0,1): no hypothesis
           [0.5, 1.0]]  # (1,0): addition, (1,1): no hypothesis
```

Args:

- `hypothesis`: A `SparseTensor` containing hypothesis sequences.
- `truth`: A `SparseTensor` containing truth sequences.
- `normalize`: A `bool`. If `True`, normalizes the Levenshtein distance by length of `truth`.
- `name`: A name for the operation (optional).

Returns:

A dense `Tensor` with rank `R - 1`, where R is the rank of the `SparseTensor` inputs `hypothesis` and `truth`.

Raises:

- `TypeError`: If either `hypothesis` or `truth` are not a `SparseTensor`.

`tf.invert_permutation(x, name=None)`

Computes the inverse permutation of a tensor.

This operation computes the inverse of an index permutation. It takes a 1-D integer tensor `x`, which represents the indices of a zero-based array, and swaps each value with its index position. In other words, for an output tensor `y`and an input tensor `x`, this operation computes the following:
`y[x[i]] = i for i in [0, 1, ..., len(x) - 1]`

The values must include 0. There can be no duplicate values or negative values.

For example:

```
# tensor `x` is [3, 4, 0, 2, 1]
invert_permutation(x) ==> [2, 4, 3, 0, 1]
```

Args:

- `x`: A `Tensor` of type `int32`. 1-D.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`. 1-D.

Other Functions and Classes

`tf.scalar_mul(scalar, x)`

Multiplies a scalar times a `Tensor` or `IndexedSlices` object.

Intended for use in gradient code which might deal with `IndexedSlices` objects, which are easy to multiply by a scalar but more expensive to multiply with arbitrary tensors.

Args:

- `scalar`: A 0-D scalar `Tensor`. Must have known shape.
- `x`: A `Tensor` or `IndexedSlices` to be scaled.

Returns:

`scalar * x` of the same type (`Tensor` or `IndexedSlices`) as `x`.

Raises:

- `ValueError`: if scalar is not a 0-D `scalar`.

`tf.sparse_segment_sqrt_n_grad(grad, indices, segment_ids, output_dim0, name=None)`

Computes gradients for SparseSegmentSqrtN.

Returns tensor "output" with same shape as grad, except for dimension 0 whose value is output_dim0.

Args:

- `grad`: A `Tensor`. Must be one of the following types: `float32`, `float64`. gradient propagated to the SparseSegmentSqrtN op.
- `indices`: A `Tensor` of type `int32`. indices passed to the corresponding SparseSegmentSqrtN op.
- `segment_ids`: A `Tensor` of type `int32`. segment_ids passed to the corresponding SparseSegmentSqrtN op.
- `output_dim0`: A `Tensor` of type `int32`. dimension 0 of "data" passed to SparseSegmentSqrtN op.
- `name`: A name for the operation (optional).

Returns:

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

Control Flow

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

Contents

- `tf.count_up_to(ref, limit, name=None)`
- `tf.cond(pred, fn1, fn2, name=None)`
- Logical Operators
- `tf.logical_and(x, y, name=None)`
- `tf.logical_not(x, name=None)`
- `tf.logical_or(x, y, name=None)`
- `tf.logical_xor(x, y, name=LogicalXor)`
- Comparison Operators
- `tf.equal(x, y, name=None)`
- `tf.not_equal(x, y, name=None)`
- `tf.less(x, y, name=None)`
- `tf.less_equal(x, y, name=None)`
- `tf.greater(x, y, name=None)`
- `tf.greater_equal(x, y, name=None)`
- `tf.select(condition, t, e, name=None)`
- `tf.where(input, name=None)`
- Debugging Operations
- `tf.is_finite(x, name=None)`
- `tf.is_inf(x, name=None)`
- `tf.is_nan(x, name=None)`
- `tf.verify_tensor_all_finite(t, msg, name=None)`
- `tf.check_numerics(tensor, message, name=None)`
- `tf.add_check_numerics_ops()`
- `tf.Assert(condition, data, summarize=None, name=None)`
- `tf.Print(input_, data, message=None, first_n=None, summarize=None, name=None)`

Control Flow Operations

TensorFlow provides several operations and classes that you can use to control the execution of operations and add conditional dependencies to your graph.

`tf.identity(input, name=None)`

Return a tensor with the same shape and contents as the input tensor or value.

Args:

- `input`: A `Tensor`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.tuple(tensors, name=None, control_inputs=None)`

Group tensors together.

This creates a tuple of tensors with the same values as the `tensors` argument, except that the value of each tensor is only returned after the values of all tensors have been computed.

`control_inputs` contains additional ops that have to finish before this op finishes, but whose outputs are not returned.
This can be used as a "join" mechanism for parallel computations: all the argument tensors can be computed in parallel, but the values of any tensor returned by `tuple` are only available after all the parallel computations are done.

See also `group` and `with_dependencies`.

Args:

- `tensors`: A list of `Tensor`s or `IndexedSlices`, some entries can be `None`.
- `name`: (optional) A name to use as a `name_scope` for the operation.

* `control_inputs`: List of additional ops to finish before returning.

Returns:

Same as `tensors`.

Raises:

* `ValueError`: If `tensors` does not contain any `Tensor` or `IndexedSlices`.
* `TypeError`: If `control_inputs` is not a list of `Operation` or `Tensor` objects.

`tf.group(*inputs, **kwargs)`

Create an op that groups multiple operations.

When this op finishes, all ops in `input` have finished. This op has no output.
See also `tuple` and `with_dependencies`.

Args:

* `*inputs`: One or more tensors to group.
* `**kwargs`: Optional parameters to pass when constructing the NodeDef.
* `name`: A name for this operation (optional).

Returns:

An Operation that executes all its inputs.

Raises:

- `ValueError`: If an unknown keyword argument is provided, or if there are no inputs.

`tf.no_op(name=None)`

Does nothing. Only useful as a placeholder for control edges.

Args:

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

Returns:

The created Operation.

`tf.count_up_to(ref, limit, name=None)`

Increments 'ref' until it reaches 'limit'.

This operation outputs "ref" after the update is done. This makes it easier to chain operations that need to use the updated value.

Args:

* `ref`: A mutable `Tensor`. Must be one of the following

  types: `int32`, `int64`. Should be from a scalar`Variable` node.
* `limit`: An `int`. If incrementing ref would bring it above limit, instead generates an 'OutOfRange' error.
* `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `ref`. A copy of the input before increment. If nothing else modifies the input, the values produced will all be distinct.

`tf.cond(pred, fn1, fn2, name=None)`

Return either fn1() or fn2() based on the boolean predicate `pred`.

`fn1` and `fn2` both return lists of output tensors. `fn1` and `fn2` must have the same non-zero number and type of outputs.

Args:

* `pred`: A scalar determining whether to return the result of `fn1` or `fn2`.
* `fn1`: The function to be performed if pred is true.
* `fn2`: The function to be performed if pref is false.
* `name`: Optional name prefix for the returned tensors.

Returns:

Tensors returned by the call to either `fn1` or `fn2`. If the functions return a singleton list, the element is extracted from the list.

Raises:

- `TypeError`: if `fn1` or `fn2` is not callable.
- `ValueError`: if `fn1` and `fn2` do not return the same number of tensors, or return tensors of different types.
- `Example`:

```
x = tf.constant(2)
y = tf.constant(5)
def f1(): return tf.mul(x, 17)
def f2(): return tf.add(y, 23)
r = cond(math_ops.less(x, y), f1, f2)
# r is set to f1().
# Operations in f2 (e.g., tf.add) are not executed.
```

Logical Operators

TensorFlow provides several operations that you can use to add logical operators to your graph.

`tf.logical_and(x, y, name=None)`

Returns the truth value of x AND y element-wise.

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.

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

Returns:

A `Tensor` of type `bool`.

`tf.logical_not(x, name=None)`

Returns the truth value of NOT x element-wise.

Args:

- `x`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.logical_or(x, y, name=None)`

Returns the truth value of x OR y element-wise.

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.

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

Returns:

A `Tensor` of type `bool`.

`tf.logical_xor(x, y, name='LogicalXor')`

x ^ y = (x | y) & ~(x & y).

Comparison Operators

TensorFlow provides several operations that you can use to add comparison operators to your graph.

`tf.equal(x, y, name=None)`

Returns the truth value of (x == y) element-wise.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`,`complex64`, `quint8`, `qint8`, `qint32`, `string`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.not_equal(x, y, name=None)`

Returns the truth value of (x != y) element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`,`complex64`, `quint8`, `qint8`, `qint32`, `string`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.less(x, y, name=None)`

Returns the truth value of (x < y) element-wise.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`,`uint16`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.less_equal(x, y, name=None)`

Returns the truth value of (x <= y) element-wise.

Args:

* `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`,`uint16`.
* `y`: A `Tensor`. Must have the same type as `x`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

```
tf.greater(x, y, name=None)
```

Returns the truth value of (x > y) element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`,`uint16`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

```
tf.greater_equal(x, y, name=None)
```

Returns the truth value of (x >= y) element-wise.

Args:

- `x`: A `Tensor`. Must be one of the following

  types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`,`uint16`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.select(condition, t, e, name=None)`

Selects elements from `t` or `e`, depending on `condition`.

The `t`, and `e` tensors must all have the same shape, and the output will also have that shape. The `condition`tensor must be a scalar if `t` and `e` are scalars. If `t` and `e` are vectors or higher rank, then `condition` must be either a vector with size matching the first dimension of `t`, or must have the same shape as `t`.

The `condition` tensor acts as a mask that chooses, based on the value at each element, whether the corresponding element / row in the output should be taken from `t` (if true) or `e` (if false).

If `condition` is a vector and `t` and `e` are higher rank matrices, then it chooses which row (outer dimension) to copy from `t` and `e`.

If `condition` has the same shape as `t` and `e`, then it chooses which element to copy from `t`and `e`.

For example:

```
# 'condition' tensor is [[True,  False]
#                        [False, True]]
# 't' is [[1, 2],
#         [3, 4]]
# 'e' is [[5, 6],
#         [7, 8]]
select(condition, t, e) ==> [[1, 6],
                             [7, 4]]
```

```
# 'condition' tensor is [True, False]
# 't' is [[1, 2],
#        [3, 4]]
# 'e' is [[5, 6],
#        [7, 8]]
select(condition, t, e) ==> [[1, 2],
                             [7, 8]]
```

Args:

- `condition`: A `Tensor` of type `bool`.
- `t`: A `Tensor` which may have the same shape as `condition`.

  If `condition` is rank 1, `t` may have higher rank, but its first dimension must match the size of `condition`.
- `e`: A `Tensor` with the same type and shape as `t`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` with the same type and shape as `t` and `e`.

`tf.where(input, name=None)`

Returns locations of true values in a boolean tensor.

This operation returns the coordinates of true elements in `input`. The coordinates are returned in a 2-D tensor where the first dimension (rows) represents the number of true elements, and the second dimension (columns) represents the coordinates of the true elements.

Keep in mind, the shape of the output tensor can vary depending on how many true values there are in `input`. Indices are output in row-major order.

For example:

```
# 'input' tensor is [[True, False]
#                    [True, False]]
# 'input' has two true values, so output has two coordinates.
# 'input' has rank of 2, so coordinates have two indices.
where(input) ==> [[0, 0],
                  [1, 0]]

# `input` tensor is [[[True, False]
#                     [True, False]]
#                    [[False, True]
#                     [False, True]]
#                    [[False, False]
#                     [False, True]]]
# 'input' has 5 true values, so output has 5 coordinates.
# 'input' has rank of 3, so coordinates have three indices.
where(input) ==> [[0, 0, 0],
                  [0, 1, 0],
                  [1, 0, 1],
                  [1, 1, 1],
                  [2, 1, 1]]
```

Args:

- `input`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

Debugging Operations

TensorFlow provides several operations that you can use to validate values and debug your graph.

`tf.is_finite(x, name=None)`

Returns which elements of x are finite.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.is_inf(x, name=None)`

Returns which elements of x are Inf.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.is_nan(x, name=None)`

Returns which elements of x are NaN.

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`tf.verify_tensor_all_finite(t, msg, name=None)`

Assert that the tensor does not contain any NaN's or Inf's.

Args:

- `t`: Tensor to check.
- `msg`: Message to log on failure.
- `name`: A name for this operation (optional).

Returns:

Same tensor as `t`.

`tf.check_numerics(tensor, message, name=None)`

Checks a tensor for NaN and Inf values.

When run, reports an `InvalidArgument` error if `tensor` has any values that are not a number (NaN) or infinity (Inf). Otherwise, passes `tensor` as-is.

Args:

- `tensor`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `message`: A `string`. Prefix of the error message.
- `name`: A name for the operation (optional).

Returns:

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

`tf.add_check_numerics_ops()`

Connect a `check_numerics` to every floating point tensor.

`check_numerics` operations themselves are added for each `float` or `double` tensor in the graph. For all ops in the graph, the `check_numerics` op for all of its (`float` or `double`) inputs is guaranteed to run before the`check_numerics` op on any of its outputs.

Returns:

A `group` op depending on all `check_numerics` ops added.

`tf.Assert(condition, data, summarize=None, name=None)`

Asserts that the given condition is true.

If `condition` evaluates to false, print the list of tensors

in `data`. `summarize` determines how many entries of the tensors to print.

Args:

- `condition`: The condition to evaluate.
- `data`: The tensors to print out when condition is false.
- `summarize`: Print this many entries of each tensor.
- `name`: A name for this operation (optional).

```
tf.Print(input_, data, message=None, first_n=None,
summarize=None, name=None)
```

Prints a list of tensors.

This is an identity op with the side effect of printing `data` when evaluating.

Args:

- `input_`: A tensor passed through this op.
- `data`: A list of tensors to print out when op is evaluated.
- `message`: A string, prefix of the error message.
- `first_n`: Only log `first_n` number of times. Negative numbers log always; this is the default.
- `summarize`: Only print this many entries of each tensor. If None, then a maximum of 3 elements are printed per input tensor.
- `name`: A name for the operation (optional).

Returns:

Same tensor as `input_`.

Images

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

Contents

chroma_downsampling=None, density_unit=None, x_density=None, y_density=None, xmp_metadata=None, name=None)

- tf.image.decode_png(contents, channels=None, dtype=None, name=None)
- tf.image.encode_png(image, compression=None, name=None)
- Resizing
- tf.image.resize_images(images, new_height, new_width, method=0, align_corners=False)
- tf.image.resize_area(images, size, align_corners=None, name=None)
- tf.image.resize_bicubic(images, size, align_corners=None, name=None)
- tf.image.resize_bilinear(images, size, align_corners=None, name=None)
- tf.image.resize_nearest_neighbor(images, size, align_corners=None, name=None)
- Cropping
- tf.image.resize_image_with_crop_or_pad(image, target_height, target_width)
- tf.image.pad_to_bounding_box(image, offset_height, offset_width, target_height, target_width)
- tf.image.crop_to_bounding_box(image, offset_height, offset_width, target_height, target_width)
- tf.image.extract_glimpse(input, size, offsets, centered=None, normalized=None, uniform_noise=None, name=None)
- Flipping and Transposing
- tf.image.flip_up_down(image)
- tf.image.random_flip_up_down(image, seed=None)
- tf.image.flip_left_right(image)
- tf.image.random_flip_left_right(image, seed=None)
- tf.image.transpose_image(image)
- Converting Between Colorspaces.
- tf.image.rgb_to_grayscale(images)
- tf.image.grayscale_to_rgb(images)
- tf.image.hsv_to_rgb(images, name=None)
- tf.image.rgb_to_hsv(images, name=None)
- tf.image.convert_image_dtype(image, dtype, saturate=False, name=None)
- Image Adjustments
- tf.image.adjust_brightness(image, delta)
- tf.image.random_brightness(image, max_delta, seed=None)
- tf.image.adjust_contrast(images, contrast_factor)

- `tf.image.random_contrast(image, lower, upper, seed=None)`
- `tf.image.adjust_hue(image, delta, name=None)`
- `tf.image.random_hue(image, max_delta, seed=None)`
- `tf.image.adjust_saturation(image, saturation_factor, name=None)`
- `tf.image.random_saturation(image, lower, upper, seed=None)`
- `tf.image.per_image_whitening(image)`
- Working with Bounding Boxes
- `tf.image.draw_bounding_boxes(images, boxes, name=None)`
- `tf.image.sample_distorted_bounding_box(image_size, bounding_boxes, seed=None, seed2=None, min_object_covered=None, aspect_ratio_range=None, area_range=None, max_attempts=None, use_image_if_no_bounding_boxes=None, name=None)`
- Other Functions and Classes
- `tf.image.saturate_cast(image, dtype)`

Encoding and Decoding

TensorFlow provides Ops to decode and encode JPEG and PNG formats. Encoded images are represented by scalar string Tensors, decoded images by 3-D uint8 tensors of shape `[height, width, channels]`. (PNG also supports uint16.)

The encode and decode Ops apply to one image at a time. Their input and output are all of variable size. If you need fixed size images, pass the output of the decode Ops to one of the cropping and resizing Ops.

Note: The PNG encode and decode Ops support RGBA, but the conversions Ops presently only support RGB, HSV, and GrayScale. Presently, the alpha channel has to be stripped from the image and re-attached using slicing ops.

```
tf.image.decode_jpeg(contents, channels=None, ratio=None,
fancy_upscaling=None, try_recover_truncated=None,
acceptable_fraction=None, name=None)
```

Decode a JPEG-encoded image to a uint8 tensor.

The attr `channels` indicates the desired number of color channels for the decoded image.

Accepted values are:

- 0: Use the number of channels in the JPEG-encoded image.
- 1: output a grayscale image.
- 3: output an RGB image.

If needed, the JPEG-encoded image is transformed to match the requested number of color channels.

The attr `ratio` allows downscaling the image by an integer factor during decoding. Allowed values are: 1, 2, 4, and 8. This is much faster than downscaling the image later.

Args:

- `contents`: A `Tensor` of type `string`. 0-D. The JPEG-encoded image.
- `channels`: An optional `int`. Defaults to `0`. Number of color channels for the decoded image.
- `ratio`: An optional `int`. Defaults to `1`. Downscaling ratio.
- `fancy_upscaling`: An optional `bool`. Defaults to `True`. If true use a slower but nicer upscaling of the chroma planes (yuv420/422 only).
- `try_recover_truncated`: An optional `bool`. Defaults to `False`. If true try to recover an image from truncated input.

- `acceptable_fraction`: An optional `float`. Defaults to `1`. The minimum required fraction of lines before a truncated input is accepted.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `uint8`. 3-D with shape `[height, width, channels]`..

```
tf.image.encode_jpeg(image, format=None, quality=None,
progressive=None, optimize_size=None,
chroma_downsampling=None, density_unit=None,
x_density=None, y_density=None, xmp_metadata=None,
name=None)
```

JPEG-encode an image.

`image` is a 3-D uint8 Tensor of shape `[height, width, channels]`.

The attr `format` can be used to override the color format of the encoded output. Values can be:

- `''`: Use a default format based on the number of channels in the image.
- `grayscale`: Output a grayscale JPEG image.

  The `channels` dimension of `image` must be 1.
- `rgb`: Output an RGB JPEG image. The `channels` dimension of `image` must be 3.

If `format` is not specified or is the empty string, a default format is picked in function of the number of channels in`image`:

- 1: Output a grayscale image.
- 3: Output an RGB image.

Args:

- `image`: A `Tensor` of type `uint8`. 3-D with shape `[height, width, channels]`.
- `format`: An optional `string` from: `"", "grayscale", "rgb"`. Defaults to `""`. Per pixel image format.
- `quality`: An optional `int`. Defaults to `95`. Quality of the compression from 0 to 100 (higher is better and slower).
- `progressive`: An optional `bool`. Defaults to `False`. If True, create a JPEG that loads progressively (coarse to fine).
- `optimize_size`: An optional `bool`. Defaults to `False`. If True, spend CPU/RAM to reduce size with no quality change.
- `chroma_downsampling`: An optional `bool`. Defaults to `True`. Seehttp://en.wikipedia.org/wiki/Chroma_subsampling.
- `density_unit`: An optional `string` from: `"in", "cm"`. Defaults to `"in"`. Unit used to specify`x_density` and `y_density`: pixels per inch (`'in'`) or centimeter (`'cm'`).
- `x_density`: An optional `int`. Defaults to `300`. Horizontal pixels per density unit.
- `y_density`: An optional `int`. Defaults to `300`. Vertical pixels per density unit.

- `xmp_metadata`: An optional `string`. Defaults to `""`. If not empty, embed this XMP metadata in the image header.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `string`. 0-D. JPEG-encoded image.

```
tf.image.decode_png(contents, channels=None, dtype=None, name=None)
```

Decode a PNG-encoded image to a uint8 or uint16 tensor.

The attr `channels` indicates the desired number of color channels for the decoded image.

Accepted values are:

- 0: Use the number of channels in the PNG-encoded image.
- 1: output a grayscale image.
- 3: output an RGB image.
- 4: output an RGBA image.

If needed, the PNG-encoded image is transformed to match the requested number of color channels.

Args:

- `contents`: A `Tensor` of type `string`. 0-D. The PNG-encoded image.

- `channels`: An optional `int`. Defaults to `0`. Number of color channels for the decoded image.
- `dtype`: An optional `tf.DType` from: `tf.uint8, tf.uint16`. Defaults to `tf.uint8`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `dtype`. 3-D with shape `[height, width, channels]`.

`tf.image.encode_png(image, compression=None, name=None)`

PNG-encode an image.

`image` is a 3-D uint8 or uint16 Tensor of shape `[height, width, channels]` where `channels` is:

- 1: for grayscale.
- 3: for RGB.
- 4: for RGBA.

The ZLIB compression level, `compression`, can be -1 for the PNG-encoder default or a value from 0 to 9. 9 is the highest compression level, generating the smallest output, but is slower.

Args:

* `image`: A `Tensor`. Must be one of the following types: `uint8`, `uint16`. 3-D with shape `[height, width, channels]`.
* `compression`: An optional `int`. Defaults to `-1`. Compression level.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `string`. 0-D. PNG-encoded image.

Resizing

The resizing Ops accept input images as tensors of several types. They always output resized images as float32 tensors.

The convenience function `resize_images()` supports both 4-D and 3-D tensors as input and output. 4-D tensors are for batches of images, 3-D tensors for individual images.
Other resizing Ops only support 4-D batches of images as input: `resize_area`, `resize_bicubic`,`resize_bilinear`, `resize_nearest_neighbor`.

Example:

```
# Decode a JPG image and resize it to 299 by 299 using default method.
image = tf.image.decode_jpeg(...)
resized_image = tf.image.resize_images(image, 299, 299)
```

`tf.image.resize_images(images, new_height, new_width, method=0, align_corners=False)`

Resize `images` to `new_width`, `new_height` using the specified `method`. Resized images will be distorted if their original aspect ratio is not the same as `new_width`, `new_height`. To avoid distortions

see `resize_image_with_crop_or_pad`.

`method` can be one of:

- `ResizeMethod.BILINEAR`: Bilinear interpolation.
- `ResizeMethod.NEAREST_NEIGHBOR`: Nearest neighbor interpolation.
- `ResizeMethod.BICUBIC`: Bicubic interpolation.
- `ResizeMethod.AREA`: Area interpolation.

Args:

- `images`: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- `new_height`: integer.
- `new_width`: integer.
- `method`: ResizeMethod. Defaults to `ResizeMethod.BILINEAR`.
- `align_corners`: bool. If true, exactly align all 4 cornets of the input and output. Defaults to `false`.

Raises:

- `ValueError`: if the shape of `images` is incompatible with the shape arguments to this function
- `ValueError`: if an unsupported resize method is specified.

Returns:

If `images` was 4-D, a 4-D float Tensor of shape `[batch, new_height, new_width, channels]`. If`images` was 3-D, a 3-D float Tensor of shape `[new_height, new_width, channels]`.

`tf.image.resize_area(images, size, align_corners=None, name=None)`

Resize `images` to `size` using area interpolation.

Input images can be of different types but output images are always float.

Args:

- `images`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `float32`,`float64`. 4-D with shape `[batch, height, width, channels]`.
- `size`: A 1-D int32 Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners`: An optional `bool`. Defaults to `False`. If true, rescale input by (new_height - 1) / (height - 1), which exactly aligns the 4 corners of images and resized images. If false, rescale by new_height / height. Treat similarly the width dimension.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. 4-D with shape `[batch, new_height, new_width, channels]`.

`tf.image.resize_bicubic(images, size, align_corners=None, name=None)`

Resize `images` to `size` using bicubic interpolation.

Input images can be of different types but output images are always float.

Args:

- `images`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `float32`,`float64`. 4-D with shape `[batch, height, width, channels]`.
- `size`: A 1-D int32 Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners`: An optional `bool`. Defaults to `False`. If true, rescale input by (new_height - 1) / (height - 1), which exactly aligns the 4 corners of images and resized images. If false, rescale by new_height / height. Treat similarly the width dimension.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. 4-D with shape `[batch, new_height, new_width, channels]`.

```
tf.image.resize_bilinear(images, size,
align_corners=None, name=None)
```

Resize `images` to `size` using bilinear interpolation.

Input images can be of different types but output images are always float.

Args:

- `images`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `float32`,`float64`. 4-D with shape `[batch, height, width, channels]`.
- `size`: A 1-D int32 Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners`: An optional `bool`. Defaults to `False`. If true, rescale input by (new_height - 1) / (height - 1), which exactly aligns the 4 corners of images and resized images. If false, rescale by new_height / height. Treat similarly the width dimension.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. 4-D with shape `[batch, new_height, new_width, channels]`.

```
tf.image.resize_nearest_neighbor(images, size,
align_corners=None, name=None)
```

Resize `images` to `size` using nearest neighbor interpolation.

Args:

- `images`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `float32`,`float64`. 4-D with shape `[batch, height, width, channels]`.
- `size`: A 1-D int32 Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners`: An optional `bool`. Defaults to `False`. If true, rescale input by (new_height - 1) / (height - 1), which exactly aligns the 4 corners of images and resized images. If false, rescale by new_height / height. Treat similarly the width dimension.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `images`. 4-D with shape `[batch, new_height, new_width, channels]`.

Cropping

```
tf.image.resize_image_with_crop_or_pad(image,
target_height, target_width)
```

Crops and/or pads an image to a target width and height.

Resizes an image to a target width and height by either centrally cropping the image or padding it evenly with zeros.

If `width` or `height` is greater than the specified `target_width` or `target_height` respectively, this op centrally crops along that dimension. If `width` or `height` is smaller than the specified `target_width` or`target_height` respectively, this op centrally pads with 0 along that dimension.

Args:

- `image`: 3-D tensor of shape [height, width, channels]
- `target_height`: Target height.
- `target_width`: Target width.

Raises:

- `ValueError`: if `target_height` or `target_width` are zero or negative.

Returns:

Cropped and/or padded image of shape `[target_height, target_width, channels]`

```
tf.image.pad_to_bounding_box(image, offset_height,
offset_width, target_height, target_width)
```

Pad `image` with zeros to the specified `height` and `width`.

Adds `offset_height` rows of zeros on top, `offset_width` columns of zeros on the left, and then pads the image on the bottom and right with zeros until it has dimensions `target_height`, `target_width`.

This op does nothing if `offset_*` is zero and the image already has size `target_height` by `target_width`.

Args:

- `image`: 3-D tensor with shape `[height, width, channels]`
- `offset_height`: Number of rows of zeros to add on top.
- `offset_width`: Number of columns of zeros to add on the left.
- `target_height`: Height of output image.
- `target_width`: Width of output image.

Returns:

3-D tensor of shape `[target_height, target_width, channels]`

Raises:

- `ValueError`: If the shape of `image` is incompatible with the `offset_*` or `target_*` arguments

---

```
tf.image.crop_to_bounding_box(image, offset_height,
offset_width, target_height, target_width)
```

Crops an image to a specified bounding box.

This op cuts a rectangular part out of `image`. The top-left corner of the returned image is at `offset_height, offset_width` in `image`, and its lower-right corner is at `offset_height + target_height, offset_width + target_width`.

Args:

- `image`: 3-D tensor with shape `[height, width, channels]`
- `offset_height`: Vertical coordinate of the top-left corner of the result in the input.
- `offset_width`: Horizontal coordinate of the top-left corner of the result in the input.
- `target_height`: Height of the result.
- `target_width`: Width of the result.

Returns:

3-D tensor of image with shape `[target_height, target_width, channels]`

Raises:

- `ValueError`: If the shape of `image` is incompatible with the `offset_*` or `target_*` arguments

---

`tf.image.extract_glimpse(input, size, offsets, centered=None, normalized=None, uniform_noise=None, name=None)`

Extracts a glimpse from the input tensor.

Returns a set of windows called glimpses extracted at location `offsets` from the input tensor. If the windows only partially overlaps the inputs, the non overlapping areas will be filled with random noise.

The result is a 4-D tensor of shape `[batch_size, glimpse_height, glimpse_width, channels]`. The channels and batch dimensions are the same as that of the input tensor. The height and width of the output windows are specified in the `size` parameter.

The argument `normalized` and `centered` controls how the windows are built: * If the coordinates are normalized but not centered, 0.0 and 1.0 correspond to the minimum and maximum of each height and width dimension. * If the coordinates are both normalized and centered, they range from -1.0 to 1.0. The coordinates (-1.0, -1.0) correspond to the upper left corner, the lower right corner is located at (1.0, 1.0) and the center is at (0, 0). * If the coordinates are not normalized they are interpreted as numbers of pixels.

Args:

- `input`: A `Tensor` of type `float32`. A 4-D float tensor of shape `[batch_size, height, width, channels]`.
- `size`: A `Tensor` of type `int32`. A 1-D tensor of 2 elements containing the size of the glimpses to extract. The glimpse height must be specified first, following by the glimpse width.
- `offsets`: A `Tensor` of type `float32`. A 2-D integer tensor of shape `[batch_size, 2]` containing the x, y locations of the center of each window.

* `centered`: An optional `bool`. Defaults to `True`. indicates if the offset coordinates are centered relative to the image, in which case the (0, 0) offset is relative to the center of the input images. If false, the (0,0) offset corresponds to the upper left corner of the input images.
* `normalized`: An optional `bool`. Defaults to `True`. indicates if the offset coordinates are normalized.
* `uniform_noise`: An optional `bool`. Defaults to `True`. indicates if the noise should be generated using a uniform distribution or a gaussian distribution.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. A tensor representing the glimpses `[batch_size, glimpse_height, glimpse_width, channels]`.

Flipping and Transposing

`tf.image.flip_up_down(image)`

Flip an image horizontally (upside down).

Outputs the contents of `image` flipped along the first dimension, which is `height`.

See also `reverse()`.

Args:

- `image`: A 3-D tensor of shape `[height, width, channels]`.

Returns:

A 3-D tensor of the same type and shape as `image`.

Raises:

- `ValueError`: if the shape of `image` not supported.

`tf.image.random_flip_up_down(image, seed=None)`

Randomly flips an image vertically (upside down).

With a 1 in 2 chance, outputs the contents of `image` flipped along the first dimension, which is `height`. Otherwise output the image as-is.

Args:

- `image`: A 3-D tensor of shape `[height, width, channels]`.
- `seed`: A Python integer. Used to create a random seed. See `set_random_seed` for behavior.

Returns:

A 3-D tensor of the same type and shape as `image`.

Raises:

- `ValueError`: if the shape of `image` not supported.

`tf.image.flip_left_right(image)`

Flip an image horizontally (left to right).

Outputs the contents of `image` flipped along the second dimension, which is `width`.

See also `reverse()`.

Args:

- `image`: A 3-D tensor of shape `[height, width, channels].`

Returns:

A 3-D tensor of the same type and shape as `image`.

Raises:

- `ValueError`: if the shape of `image` not supported.

`tf.image.random_flip_left_right(image, seed=None)`

Randomly flip an image horizontally (left to right).

With a 1 in 2 chance, outputs the contents of `image` flipped along the second dimension, which is `width`. Otherwise output the image as-is.

Args:

- `image`: A 3-D tensor of shape `[height, width, channels].`
- `seed`: A Python integer. Used to create a random seed.

  See `set_random_seed` for behavior.

Returns:

A 3-D tensor of the same type and shape as `image`.

Raises:

- `ValueError`: if the shape of `image` not supported.

`tf.image.transpose_image(image)`

Transpose an image by swapping the first and second dimension.

See also `transpose()`.

Args:

- `image`: 3-D tensor of shape `[height, width, channels]`

Returns:

A 3-D tensor of shape `[width, height, channels]`

Raises:

- `ValueError`: if the shape of `image` not supported.

Converting Between Colorspaces.

Image ops work either on individual images or on batches of images, depending on the shape of their input Tensor.

If 3-D, the shape is `[height, width, channels]`, and the Tensor represents one image. If 4-D, the shape is `[batch_size, height, width, channels]`, and the Tensor represents `batch_size` images.

Currently, `channels` can usefully be 1, 2, 3, or 4. Single-channel images are grayscale, images with 3 channels are encoded as either RGB or HSV. Images with 2 or 4 channels include an alpha channel, which has to be stripped from the image before passing the image to most image processing functions (and can be re-attached later).

Internally, images are either stored in as one `float32` per channel per pixel (implicitly, values are assumed to lie in `[0,1)`) or one `uint8` per channel per pixel (values are assumed to lie in `[0,255]`).

Tensorflow can convert between images in RGB or HSV. The conversion functions work only on float images, so you need to convert images in other formats using `convert_image_dtype`.

Example:

```
# Decode an image and convert it to HSV.
rgb_image = tf.decode_png(...,  channels=3)
rgb_image_float = tf.convert_image_dtype(rgb_image, tf.float32)
hsv_image = tf.rgb_to_hsv(rgb_image)
```

`tf.image.rgb_to_grayscale(images)`

Converts one or more images from RGB to Grayscale.

Outputs a tensor of the same `DType` and rank as `images`. The size of the last dimension of the output is 1, containing the Grayscale value of the pixels.

Args:

- `images`: The RGB tensor to convert. Last dimension must have size 3 and should contain RGB values.

Returns:

The converted grayscale image(s).

`tf.image.grayscale_to_rgb(images)`

Converts one or more images from Grayscale to RGB.

Outputs a tensor of the same `DType` and rank as `images`. The size of the last dimension of the output is 3, containing the RGB value of the pixels.

Args:

* `images`: The Grayscale tensor to convert. Last dimension must be size 1.

Returns:

The converted grayscale image(s).

```
tf.image.hsv_to_rgb(images, name=None)
```

Convert one or more images from HSV to RGB.

Outputs a tensor of the same shape as the `images` tensor, containing the RGB value of the pixels. The output is only well defined if the value in `images` are in `[0,1]`.

See `rgb_to_hsv` for a description of the HSV encoding.

Args:

* `images`: A `Tensor` of type `float32`. 1-D or higher rank. HSV data to convert. Last dimension must be size 3.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. `images` converted to RGB.

```
tf.image.rgb_to_hsv(images, name=None)
```

Converts one or more images from RGB to HSV.

Outputs a tensor of the same shape as the `images` tensor, containing the HSV value of the pixels. The output is only well defined if the value in `images` are in `[0,1]`.

`output[..., 0]` contains hue, `output[..., 1]` contains saturation, and `output[..., 2]` contains value. All HSV values are in `[0,1]`. A hue of 0 corresponds to pure red, hue 1/3 is pure green, and 2/3 is pure blue.

Args:

- `images`: A `Tensor` of type `float32`. 1-D or higher rank. RGB data to convert. Last dimension must be size 3.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. `images` converted to HSV.

```
tf.image.convert_image_dtype(image, dtype,
saturate=False, name=None)
```

Convert `image` to `dtype`, scaling its values if needed.

Images that are represented using floating point values are expected to have values in the range [0,1). Image data stored in integer data types are expected to have values in the range `[0,MAX]`, where `MAX` is the largest positive representable number for the data type.

This op converts between data types, scaling the values appropriately before casting.

Note that converting from floating point inputs to integer types may lead to over/underflow problems. Set saturate to `True` to avoid such problem in problematic conversions. If enabled, saturation will clip the output into the allowed range before performing a potentially dangerous cast (and only before performing such a cast, i.e., when casting from a floating point to an integer type, and when casting from a signed to an unsigned type; `saturate`has no effect on casts between floats, or on casts that increase the type's range).

Args:

- `image`: An image.
- `dtype`: A `DType` to convert `image` to.
- `saturate`: If `True`, clip the input before casting (if necessary).
- `name`: A name for this operation (optional).

Returns:

`image`, converted to `dtype`.

Image Adjustments

TensorFlow provides functions to adjust images in various ways: brightness, contrast, hue, and saturation. Each adjustment can be done with predefined parameters or with random parameters picked from predefined intervals. Random adjustments are often useful to expand a training set and reduce overfitting.

If several adjustments are chained it is advisable to minimize the number of redundant conversions by first converting the images to the most natural data type and representation (RGB or HSV).

`tf.image.adjust_brightness(image, delta)`

Adjust the brightness of RGB or Grayscale images.

This is a convenience method that converts an RGB image to float representation, adjusts its brightness, and then converts it back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

The value `delta` is added to all components of the tensor `image`.

Both `image` and `delta` are converted to`float` before adding

(and `image` is scaled appropriately if it is in fixed-point

representation). For regular images,`delta` should be in the

range `[0,1)`, as it is added to the image in floating point

representation, where pixel values are in the `[0,1)` range.

Args:

- `image`: A tensor.
- `delta`: A scalar. Amount to add to the pixel values.

Returns:

A brightness-adjusted tensor of the same shape and type as `image`.

`tf.image.random_brightness(image, max_delta, seed=None)`

Adjust the brightness of images by a random factor.

Equivalent to `adjust_brightness()` using a `delta` randomly picked in the interval `[-max_delta, max_delta)`.

Args:

- `image`: An image.
- `max_delta`: float, must be non-negative.
- `seed`: A Python integer. Used to create a random seed.

  See `set_random_seed` for behavior.

Returns:

The brightness-adjusted image.

Raises:

- `ValueError`: if `max_delta` is negative.

`tf.image.adjust_contrast(images, contrast_factor)`

Adjust contrast of RGB or grayscale images.

This is a convenience method that converts an RGB image to float representation, adjusts its contrast, and then converts it back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`images` is a tensor of at least 3 dimensions. The last 3 dimensions are interpreted as `[height, width, channels]`. The other dimensions only represent a collection of images, such as `[batch, height, width, channels].`

Contrast is adjusted independently for each channel of each image.

For each channel, this Op computes the mean of the image pixels in the channel and then adjusts each component `x` of each pixel to `(x - mean) * contrast_factor + mean`.

Args:

- `images`: Images to adjust. At least 3-D.
- `contrast_factor`: A float multiplier for adjusting contrast.

Returns:

The constrast-adjusted image or images.

`tf.image.random_contrast(image, lower, upper, seed=None)`

Adjust the contrast of an image by a random factor.

Equivalent to `adjust_contrast()` but uses a `contrast_factor` randomly picked in the interval `[lower, upper]`.

Args:

- `image`: An image tensor with 3 or more dimensions.
- `lower`: float. Lower bound for the random contrast factor.
- `upper`: float. Upper bound for the random contrast factor.
- `seed`: A Python integer. Used to create a random seed.

  See `set_random_seed` for behavior.

Returns:

The contrast-adjusted tensor.

Raises:

- `ValueError`: if `upper <= lower` or if `lower < 0`.

`tf.image.adjust_hue(image, delta, name=None)`

Adjust hue of an RGB image.

This is a convenience method that converts an RGB image to float representation, converts it to HSV, add an offset to the hue channel, converts back to RGB and then back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`image` is an RGB image. The image hue is adjusted by converting the image to HSV and rotating the hue channel (H) by `delta`. The image is then converted back to RGB.

`delta` must be in the interval `[-1, 1]`.

Args:

- `image`: RGB image or images. Size of the last dimension must be 3.
- `delta`: float. How much to add to the hue channel.
- `name`: A name for this operation (optional).

Returns:

Adjusted image(s), same shape and DType as `image`.

`tf.image.random_hue(image, max_delta, seed=None)`

Adjust the hue of an RGB image by a random factor.

Equivalent to `adjust_hue()` but uses a `delta` randomly picked in the interval `[-max_delta, max_delta]`.

`max_delta` must be in the interval `[0, 0.5]`.

Args:

- `image`: RGB image or images. Size of the last dimension must be 3.
- `max_delta`: float. Maximum value for the random delta.
- `seed`: An operation-specific seed. It will be used in conjunction with the graph-level seed to determine the real seeds that will be used in this operation. Please see the documentation of set_random_seed for its interaction with the graph-level random seed.

Returns:

3-D float tensor of shape `[height, width, channels]`.

Raises:

- `ValueError`: if `max_delta` is invalid.

`tf.image.adjust_saturation(image, saturation_factor, name=None)`

Adjust saturation of an RGB image.

This is a convenience method that converts an RGB image to float representation, converts it to HSV, add an offset to the saturation channel, converts back to RGB and then back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`image` is an RGB image. The image saturation is adjusted by converting the image to HSV and multiplying the saturation (S) channel by `saturation_factor` and clipping. The image is then converted back to RGB.

Args:

- `image`: RGB image or images. Size of the last dimension must be 3.
- `saturation_factor`: float. Factor to multiply the saturation by.
- `name`: A name for this operation (optional).

Returns:

Adjusted image(s), same shape and DType as `image`.

`tf.image.random_saturation(image, lower, upper, seed=None)`

Adjust the saturation of an RGB image by a random factor.

Equivalent to `adjust_saturation()` but uses

a `saturation_factor` randomly picked in the interval `[lower,`

`upper]`.

Args:

- `image`: RGB image or images. Size of the last dimension must be 3.
- `lower`: float. Lower bound for the random saturation factor.
- `upper`: float. Upper bound for the random saturation factor.
- `seed`: An operation-specific seed. It will be used in conjunction with

the graph-level seed to determine the real seeds that will be used in this operation. Please see the documentation of set_random_seed for its interaction with the graph-level random seed.

Returns:

Adjusted image(s), same shape and DType as `image`.

Raises:

- `ValueError`: if `upper <= lower` or if `lower < 0`.

`tf.image.per_image_whitening(image)`

Linearly scales `image` to have zero mean and unit norm.

This op computes `(x - mean) / adjusted_stddev`, where `mean` is the average of all values in image, and`adjusted_stddev = max(stddev, 1.0/sqrt(image.NumElements()))`.

`stddev` is the standard deviation of all values in `image`. It is capped away from zero to protect against division by 0 when handling uniform images.

Note that this implementation is limited: * It only whitens based on the statistics of an individual image. * It does not take into account the covariance structure.

Args:

- `image`: 3-D tensor of shape `[height, width, channels]`.

Returns:

The whitened image with same shape as `image`.

Raises:

- `ValueError`: if the shape of 'image' is incompatible with this function.

Working with Bounding Boxes

`tf.image.draw_bounding_boxes(images, boxes, name=None)`

Draw bounding boxes on a batch of images.

Outputs a copy of `images` but draws on top of the pixels zero or more bounding boxes specified by the locations in `boxes`. The coordinates of the each bounding box in `boxes are encoded as`[y_min, x_min, y_max, x_max]. `The bounding box coordinates are floats in`[0.0, 1.0]` relative to the width and height of the underlying image. For example, if an image is 100 x 200 pixels and the bounding box is `[0.1, 0.5, 0.2, 0.9]`, the bottom-left and upper-right coordinates of the bounding box will be `(10, 40)` to `(50, 180)`.

Parts of the bounding box may fall outside the image.

Args:

- `images`: A `Tensor` of type `float32`. 4-D with shape `[batch, height, width, depth]`. A batch of images.
- `boxes`: A `Tensor` of type `float32`. 3-D with shape `[batch, num_bounding_boxes, 4]` containing bounding boxes.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`. 4-D with the same shape as `images`. The batch of input images with bounding boxes drawn on the images.

`tf.image.sample_distorted_bounding_box(image_size, bounding_boxes, seed=None, seed2=None, min_object_covered=None, aspect_ratio_range=None, area_range=None, max_attempts=None, use_image_if_no_bounding_boxes=None, name=None)`

Generate a single randomly distorted bounding box for an image.

Bounding box annotations are often supplied in addition to ground-truth labels in image recognition or object localization tasks. A common technique for training such a system is to randomly distort an image while preserving its content, i.e. data augmentation. This Op outputs a randomly distorted localization of an object, i.e. bounding box, given an `image_size`, `bounding_boxes` and a series of constraints.
The output of this Op is a single bounding box that may be used to crop the original image. The output is returned as 3 tensors: `begin`, `size` and `bboxes`. The first 2 tensors can be fed directly into `tf.slice` to crop the image. The latter may be supplied to `tf.image.draw_bounding_box` to visualize what the bounding box looks like.
Bounding boxes are supplied and returned as `[y_min, x_min, y_max, x_max]`. The bounding box coordinates are floats in `[0.0, 1.0]` relative to the width and height of the underlying image.

For example,

```
# Generate a single distorted bounding box.
begin, size, bbox_for_draw =
tf.image.sample_distorted_bounding_box(
    tf.shape(image),
    bounding_boxes=bounding_boxes)

# Draw the bounding box in an image summary.
image_with_box =
tf.image.draw_bounding_boxes(tf.expand_dims(image, 0),
                                             bbox_for_draw)
tf.image_summary('images_with_box', image_with_box)
```

```
# Employ the bounding box to distort the image.
distorted_image = tf.slice(image, begin, size)
```

Note that if no bounding box information is available, setting `use_image_if_no_bounding_boxes = true`will assume there is a single implicit bounding box covering the whole image. If`use_image_if_no_bounding_boxes` is false and no bounding boxes are supplied, an error is raised.

Args:

- `image_size`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`. 1-D, containing `[height, width, channels]`.
- `bounding_boxes`: A `Tensor` of type `float32`. 3-D with shape `[batch, N, 4]` describing the N bounding boxes associated with the image.
- `seed`: An optional `int`. Defaults to `0`. If either `seed` or `seed2` are set to non-zero, the random number generator is seeded by the given `seed`. Otherwise, it is seeded by a random seed.
- `seed2`: An optional `int`. Defaults to `0`. A second seed to avoid seed collision.
- `min_object_covered`: An optional `float`. Defaults to `0.1`. The cropped area of the image must contain at least this fraction of any bounding box supplied.
- `aspect_ratio_range`: An optional list of `floats`. Defaults to `[0.75, 1.33]`. The cropped area of the image must have an aspect ratio = width / height within this range.

- `area_range`: An optional list of `floats`. Defaults to `[0.05, 1]`. The cropped area of the image must contain a fraction of the supplied image within in this range.
- `max_attempts`: An optional `int`. Defaults to `100`. Number of attempts at generating a cropped region of the image of the specified constraints. After `max_attempts` failures, return the entire image.
- `use_image_if_no_bounding_boxes`: An optional `bool`. Defaults to `False`. Controls behavior if no bounding boxes supplied. If true, assume an implicit bounding box covering the whole input. If false, raise an error.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (begin, size, bboxes).

- `begin`: A `Tensor`. Has the same type as `image_size`. 1-D, containing `[offset_height, offset_width, 0]`. Provide as input to `tf.slice`.
- `size`: A `Tensor`. Has the same type as `image_size`. 1-D, containing `[target_height, target_width, -1]`. Provide as input to `tf.slice`.
- `bboxes`: A `Tensor` of type `float32`. 3-D with shape `[1, 1, 4]` containing the distorted bounding box. Provide as input to `tf.image.draw_bounding_boxes`.

Other Functions and Classes

`tf.image.saturate_cast(image, dtype)`

Performs a safe cast of image data to `dtype`.

This function casts the data in image to `dtype`, without applying any scaling. If there is a danger that image data would over or underflow in the cast, this op applies the appropriate clamping before the cast.

Args:

- `image`: An image to cast to a different data type.
- `dtype`: A `DType` to cast `image` to.

Returns:

`image`, safely cast to `dtype`.

Sparse Tensors

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

Contents

Sparse Tensor Representation

Tensorflow supports a `SparseTensor` representation for data that is sparse in multiple dimensions. Contrast this representation with `IndexedSlices`, which is efficient for representing tensors that are sparse in their first dimension, and dense along all other dimensions.

`class tf.SparseTensor`

Represents a sparse tensor.

Tensorflow represents a sparse tensor as three separate dense tensors: `indices`, `values`, and `shape`. In Python, the three tensors are collected into a `SparseTensor` class for ease of use. If you have separate `indices`,`values`, and `shape` tensors, wrap them in a `SparseTensor` object before passing to the ops below.

Concretely, the sparse tensor `SparseTensor(indices, values, shape)` is

- `indices`: A 2-D int64 tensor of shape `[N, ndims]`.

- `values`: A 1-D tensor of any type and shape `[N]`.
- `shape`: A 1-D int64 tensor of shape `[ndims]`.

where `N` and `ndims` are the number of values, and number of dimensions in the `SparseTensor` respectively.

The corresponding dense tensor satisfies

```
dense.shape = shape
dense[tuple(indices[i])] = values[i]
```

By convention, `indices` should be sorted in row-major order (or equivalently lexicographic order on the tuples`indices[i]`). This is not enforced when `SparseTensor` objects are constructed, but most ops assume correct ordering. If the ordering of sparse tensor `st` is wrong, a fixed version can be obtained by calling`tf.sparse_reorder(st)`.

Example: The sparse tensor

```
SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], shape=[3, 4])
```

represents the dense tensor

```
[[1, 0, 0, 0]
 [0, 0, 2, 0]
 [0, 0, 0, 0]]
```

`tf.SparseTensor.__init__(indices, values, shape)`

Creates a `SparseTensor`.

Args:

- `indices`: A 2-D int64 tensor of shape `[N, ndims]`.
- `values`: A 1-D tensor of any type and shape `[N]`.
- `shape`: A 1-D int64 tensor of shape `[ndims]`.

Returns:

A `SparseTensor`

`tf.SparseTensor.indices`

The indices of non-zero values in the represented dense tensor.

Returns:

A 2-D Tensor of int64 with shape `[N, ndims]`, where `N` is the number of non-zero values in the tensor, and`ndims` is the rank.

`tf.SparseTensor.values`

The non-zero values in the represented dense tensor.

Returns:

A 1-D Tensor of any data type.

`tf.SparseTensor.dtype`

The `DType` of elements in this tensor.

`tf.SparseTensor.shape`

A 1-D Tensor of int64 representing the shape of the dense tensor.

`tf.SparseTensor.graph`

The `Graph` that contains the index, value, and shape tensors.

`class tf.SparseTensorValue`

SparseTensorValue(indices, values, shape)

`tf.SparseTensorValue.indices`

Alias for field number 0

`tf.SparseTensorValue.shape`

Alias for field number 2

`tf.SparseTensorValue.values`

Alias for field number 1

Sparse to Dense Conversion

`tf.sparse_to_dense(sparse_indices, output_shape, sparse_values, default_value=0, validate_indices=True, name=None)`

Converts a sparse representation into a dense tensor.

Builds an array `dense` with shape `output_shape` such that

```
# If sparse_indices is scalar
dense[i] = (i == sparse_indices ? sparse_values : default_value)

# If sparse_indices is a vector, then for each i
dense[sparse_indices[i]] = sparse_values[i]

# If sparse_indices is an n by d matrix, then for each i in [0,
n)
dense[sparse_indices[i][0], ..., sparse_indices[i][d-1]] =
sparse_values[i]
```

All other values in `dense` are set to `default_value`.

If `sparse_values` is a scalar, all sparse indices are set to this single value.

Indices should be sorted in lexicographic order, and indices must not contain any repeats. If `validate_indices`is True, these properties are checked during execution.

Args:

- `sparse_indices`: A 0-D, 1-D, or 2-D `Tensor` of type `int32` or `int64`. `sparse_indices[i]` contains the complete index where `sparse_values[i]` will be placed.
- `output_shape`: A 1-D `Tensor` of the same type as `sparse_indices`. Shape of the dense output tensor.
- `sparse_values`: A 0-D or 1-D `Tensor`. Values corresponding to each row of `sparse_indices`, or a scalar value to be used for all sparse indices.
- `default_value`: A 0-D `Tensor` of the same type as `sparse_values`. Value to set for indices not specified in `sparse_indices`. Defaults to zero.
- `validate_indices`: A boolean value. If True, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats.
- `name`: A name for the operation (optional).

Returns:

Dense `Tensor` of shape `output_shape`. Has the same type as `sparse_values`.

```
tf.sparse_tensor_to_dense(sp_input, default_value=0,
validate_indices=True, name=None)
```

Converts a `SparseTensor` into a dense tensor.

This op is a convenience wrapper around `sparse_to_dense` for `SparseTensor`s.

For example, if `sp_input` has shape `[3, 5]` and non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
```

and `default_value` is `x`, then the output will be a dense `[3, 5]` string tensor with values:

```
[[x a x b x]
 [x x x x x]
 [c x x x x]]
```

Indices must be without repeats. This is only tested if validate_indices is True.

Args:

- `sp_input`: The input `SparseTensor`.
- `default_value`: Scalar value to set for indices not specified in `sp_input`. Defaults to zero.
- `validate_indices`: A boolean value. If `True`, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats.
- `name`: A name prefix for the returned tensors (optional).

Returns:

A dense tensor with shape `sp_input.shape` and values specified by the non-empty values in `sp_input`. Indices not in `sp_input` are assigned `default_value`.

Raises:

- `TypeError`: If `sp_input` is not a `SparseTensor`.

`tf.sparse_to_indicator(sp_input, vocab_size, name=None)`

Converts a `SparseTensor` of ids into a dense bool indicator tensor.

The last dimension of `sp_input` is discarded and replaced with the values of `sp_input`. If `sp_input.shape = [D0, D1, ..., Dn, K]`, then `output.shape = [D0, D1, ..., Dn, vocab_size]`, where

```
output[d_0, d_1, ..., d_n, sp_input[d_0, d_1, ..., d_n, k]] =
True
```

and False elsewhere in `output`.

For example, if `sp_input.shape = [2, 3, 4]` with non-empty values:

```
[0, 0, 0]: 0
[0, 1, 0]: 10
[1, 0, 3]: 103
[1, 1, 2]: 150
[1, 1, 3]: 149
[1, 1, 4]: 150
[1, 2, 1]: 121
```

and `vocab_size = 200`, then the output will be a `[2, 3, 200]` dense bool tensor with False everywhere except at positions

```
(0, 0, 0), (0, 1, 10), (1, 0, 103), (1, 1, 149), (1, 1, 150),
(1, 2, 121).
```

Note that repeats are allowed in the input SparseTensor. This op is useful for converting `SparseTensor`s into dense formats for compatibility with ops that expect dense tensors.

The input `SparseTensor` must be in row-major order.

Args:

- `sp_input`: A `SparseTensor` of type `int32` or `int64`.
- `vocab_size`: The new size of the last dimension, with `all(0 <= sp_input.values < vocab_size)`.
- `name`: A name prefix for the returned tensors (optional)

Returns:

A dense bool indicator tensor representing the indices with specified value.

Raises:

- `TypeError`: If `sp_input` is not a `SparseTensor`.

Manipulation

`tf.sparse_concat(concat_dim, sp_inputs, name=None)`

Concatenates a list of `SparseTensor` along the specified dimension.

Concatenation is with respect to the dense versions of each sparse input. It is assumed that each inputs is a`SparseTensor` whose elements are ordered along increasing dimension number.
All inputs' shapes must match, except for the concat dimension.

The `indices`, `values`, and `shapes` lists must have the same length.

The output shape is identical to the inputs', except along the concat dimension, where it is the sum of the inputs' sizes along that dimension.

The output elements will be resorted to preserve the sort order along increasing dimension number.

This op runs in `O(M log M)` time, where `M` is the total number of non-empty values across all inputs. This is due to the need for an internal sort in order to concatenate efficiently across an arbitrary dimension.

For example, if `concat_dim = 1` and the inputs are

```
sp_inputs[0]: shape = [2, 3]
[0, 2]: "a"
[1, 0]: "b"
[1, 1]: "c"

sp_inputs[1]: shape = [2, 4]
[0, 1]: "d"
[0, 2]: "e"
```

then the output will be

```
shape = [2, 7]
[0, 2]: "a"
[0, 4]: "d"
[0, 5]: "e"
[1, 0]: "b"
[1, 1]: "c"
```

Graphically this is equivalent to doing

```
[    a] concat [  d e  ] = [    a   d e  ]
[b c  ]        [       ]   [b c          ]
```

Args:

- `concat_dim`: Dimension to concatenate along.
- `sp_inputs`: List of `SparseTensor` to concatenate.
- `name`: A name prefix for the returned tensors (optional).

Returns:

A `SparseTensor` with the concatenated output.

Raises:

- `TypeError`: If `sp_inputs` is not a list of `SparseTensor`.

`tf.sparse_reorder(sp_input, name=None)`

Reorders a `SparseTensor` into the canonical, row-major ordering.

Note that by convention, all sparse ops preserve the canonical ordering along increasing dimension number. The only time ordering can be violated is during manual manipulation of the indices and values to add entries.

Reordering does not affect the shape of the `SparseTensor`.

For example, if `sp_input` has shape `[4, 5]` and `indices` / `values`:

```
[0, 3]: b
[0, 1]: a
[3, 1]: d
[2, 0]: c
```

then the output will be a `SparseTensor` of shape `[4,`

`5]` and `indices` / `values`:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

Args:

- `sp_input`: The input `SparseTensor`.

* `name`: A name prefix for the returned tensors (optional)

Returns:

A `SparseTensor` with the same shape and non-empty values, but in canonical ordering.

Raises:

* `TypeError`: If `sp_input` is not a `SparseTensor`.

`tf.sparse_split(split_dim, num_split, sp_input, name=None)`

Split a `SparseTensor` into `num_split` tensors along `split_dim`.

If the `sp_input.shape[split_dim]` is not an integer multiple of `num_split` each slice starting from 0:`shape[split_dim] % num_split` gets extra one dimension. For example, if `split_dim = 1` and`num_split = 2` and the input is:

```
input_tensor = shape = [2, 7]
[    a   d e  ]
[b c          ]
```

Graphically the output tensors are:

```
output_tensor[0] =
[    a ]
[b c   ]

output_tensor[1] =
[ d e  ]
[      ]
```

Args:

- `split_dim`: A 0-D `int32` `Tensor`. The dimension along which to split.
- `num_split`: A Python integer. The number of ways to split.
- `sp_input`: The `SparseTensor` to split.
- `name`: A name for the operation (optional).

Returns:

`num_split` `SparseTensor` objects resulting from splitting `value`.

Raises:

- `TypeError`: If `sp_input` is not a `SparseTensor`.

`tf.sparse_retain(sp_input, to_retain)`

Retains specified non-empty values within a `SparseTensor`.

For example, if `sp_input` has shape `[4, 5]` and 4 non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

and `to_retain = [True, False, False, True]`, then the output will be a `SparseTensor` of shape `[4, 5]` with 2 non-empty values:

```
[0, 1]: a
[3, 1]: d
```

Args:

* `sp_input`: The input `SparseTensor` with `N` non-empty elements.
* `to_retain`: A bool vector of length `N` with `M` true values.

Returns:

A `SparseTensor` with the same shape as the input and `M` non-empty elements corresponding to the true positions in `to_retain`.

Raises:

* `TypeError`: If `sp_input` is not a `SparseTensor`.

`tf.sparse_fill_empty_rows(sp_input, default_value, name=None)`

Fills empty rows in the input 2-D `SparseTensor` with a default value.

This op adds entries with the specified `default_value` at

index `[row, 0]` for any row in the input that does not already have a value.

For example, suppose `sp_input` has shape `[5, 6]` and non-empty values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

Rows 1 and 4 are empty, so the output will be of shape `[5, 6]` with values:

```
[0, 1]: a
```

```
[0, 3]: b
[1, 0]: default_value
[2, 0]: c
[3, 1]: d
[4, 0]: default_value
```

Note that the input may have empty columns at the end, with no effect on this op.

The output `SparseTensor` will be in row-major order and will have the same shape as the input.

This op also returns an indicator vector such that

```
empty_row_indicator[i] = True iff row i was an empty row.
```

Args:

- `sp_input`: A `SparseTensor` with shape `[N, M]`.
- `default_value`: The value to fill for empty rows, with the same type as `sp_input.`
- `name`: A name prefix for the returned tensors (optional)

Returns:

- `sp_ordered_output`: A `SparseTensor` with shape `[N, M]`, and with all empty rows filled in with`default_value`.
- `empty_row_indicator`: A bool vector of length `N` indicating whether each input row was empty.

Raises:

- `TypeError`: If `sp_input` is not a `SparseTensor`.

Inputs and Readers

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

Contents

* `tf.train.limit_epochs(tensor, num_epochs=None, name=None)`
* `tf.train.range_input_producer(limit, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`
* `tf.train.slice_input_producer(tensor_list, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`
* `tf.train.string_input_producer(string_tensor, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`
* Batching at the end of an input pipeline
* `tf.train.batch(tensor_list, batch_size, num_threads=1, capacity=32, enqueue_many=False, shapes=None, name=None)`
* `tf.train.batch_join(tensor_list_list, batch_size, capacity=32, enqueue_many=False, shapes=None, name=None)`
* `tf.train.shuffle_batch(tensor_list, batch_size, capacity, min_after_dequeue, num_threads=1, seed=None, enqueue_many=False, shapes=None, name=None)`
* `tf.train.shuffle_batch_join(tensor_list_list, batch_size, capacity, min_after_dequeue, seed=None, enqueue_many=False, shapes=None, name=None)`

Placeholders

TensorFlow provides a placeholder operation that must be fed with data on execution. For more info, see the section on Feeding data.

`tf.placeholder(dtype, shape=None, name=None)`

Inserts a placeholder for a tensor that will be always fed.

Important: This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument

to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

For example:

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))
```

```
y = tf.matmul(x, x)

with tf.Session() as sess:
  print(sess.run(y))  # ERROR: will fail because x was not fed.

  rand_array = np.random.rand(1024, 1024)
  print(sess.run(y, feed_dict={x: rand_array}))  # Will succeed.
```

Args:

- `dtype`: The type of elements in the tensor to be fed.
- `shape`: The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` that may be used as a handle for feeding a value, but not evaluated directly.

Readers

TensorFlow provides a set of Reader classes for reading data formats. For more information on inputs and readers, see Reading data.

`class tf.ReaderBase`

Base class for different Reader types, that produce a record every step.

Conceptually, Readers convert string 'work units' into records (key, value pairs). Typically the 'work units' are filenames and the records are extracted from the contents of those files. We want a single record produced per step, but a work unit can correspond to many records.

Therefore we introduce some decoupling using a queue. The queue contains the work units and the Reader dequeues from the queue when it is asked to produce a record (via Read()) but it has finished the last work unit.

`tf.ReaderBase.__init__(reader_ref, supports_serialize=False)`

Creates a new ReaderBase.

Args:

- `reader_ref`: The operation that implements the reader.
- `supports_serialize`: True if the reader implementation can serialize its state.

`tf.ReaderBase.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.ReaderBase.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.ReaderBase.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name`: A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key`: A string scalar Tensor.
- `value`: A string scalar Tensor.

`tf.ReaderBase.reader_ref`

Op that implements the reader.

`tf.ReaderBase.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.ReaderBase.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.ReaderBase.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.ReaderBase.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.TextLineReader`

A Reader that outputs the lines of a file delimited by newlines.

Newlines are stripped from the output. See ReaderBase for supported methods.

`tf.TextLineReader.__init__(skip_header_lines=None, name=None)`

Create a TextLineReader.

Args:

- `skip_header_lines`: An optional int. Defaults to 0. Number of lines to skip from the beginning of every file.
- `name`: A name for the operation (optional).

`tf.TextLineReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.TextLineReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.TextLineReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.

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

Returns:

A tuple of Tensors (key, value).

* `key`: A string scalar Tensor.
* `value`: A string scalar Tensor.

`tf.TextLineReader.reader_ref`

Op that implements the reader.

`tf.TextLineReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.TextLineReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.TextLineReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.TextLineReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.WholeFileReader`

A Reader that outputs the entire contents of a file as a value.

To use, enqueue filenames in a Queue. The output of Read will be a filename (key) and the contents of that file (value).

See ReaderBase for supported methods.

`tf.WholeFileReader.__init__(name=None)`

Create a WholeFileReader.

Args:

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

`tf.WholeFileReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.WholeFileReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.WholeFileReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

* `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.

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

Returns:

A tuple of Tensors (key, value).

* `key`: A string scalar Tensor.
* `value`: A string scalar Tensor.

`tf.WholeFileReader.reader_ref`

Op that implements the reader.

`tf.WholeFileReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.WholeFileReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

* `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
* `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.WholeFileReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.WholeFileReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.IdentityReader`

A Reader that outputs the queued work as both the key and value.

To use, enqueue strings in a Queue. Read will take the front work string and output (work, work).

See ReaderBase for supported methods.

`tf.IdentityReader.__init__(name=None)`

Create a IdentityReader.

Args:

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

`tf.IdentityReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.IdentityReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.IdentityReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

* `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.

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

Returns:

A tuple of Tensors (key, value).

- `key`: A string scalar Tensor.
- `value`: A string scalar Tensor.

`tf.IdentityReader.reader_ref`

Op that implements the reader.

`tf.IdentityReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.IdentityReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.IdentityReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.IdentityReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.TFRecordReader`

A Reader that outputs the records from a TFRecords file.

See ReaderBase for supported methods.

`tf.TFRecordReader.__init__(name=None)`

Create a TFRecordReader.

Args:

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

`tf.TFRecordReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.TFRecordReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.TFRecordReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

* `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
* `name`: A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key`: A string scalar Tensor.
- `value`: A string scalar Tensor.

`tf.TFRecordReader.reader_ref`

Op that implements the reader.

`tf.TFRecordReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.TFRecordReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.TFRecordReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.TFRecordReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.FixedLengthRecordReader`

A Reader that outputs fixed-length records from a file.

See ReaderBase for supported methods.

`tf.FixedLengthRecordReader.__init__(record_bytes, header_bytes=None, footer_bytes=None, name=None)`

Create a FixedLengthRecordReader.

Args:

- `record_bytes`: An int.
- `header_bytes`: An optional int. Defaults to 0.
- `footer_bytes`: An optional int. Defaults to 0.
- `name`: A name for the operation (optional).

`tf.FixedLengthRecordReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

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

Returns:

An int64 Tensor.

`tf.FixedLengthRecordReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

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

Returns:

An int64 Tensor.

`tf.FixedLengthRecordReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue`: A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name`: A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key`: A string scalar Tensor.
- `value`: A string scalar Tensor.

`tf.FixedLengthRecordReader.reader_ref`

Op that implements the reader.

`tf.FixedLengthRecordReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

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

Returns:

The created Operation.

`tf.FixedLengthRecordReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state`: A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name`: A name for the operation (optional).

Returns:

The created Operation.

`tf.FixedLengthRecordReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

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

Returns:

A string Tensor.

`tf.FixedLengthRecordReader.supports_serialize`

Whether the Reader implementation can serialize its state.

Converting

TensorFlow provides several operations that you can use to convert various data formats into tensors.

`tf.decode_csv(records, record_defaults, field_delim=None, name=None)`

Convert CSV records to tensors. Each column maps to one tensor.

RFC 4180 format is expected for the CSV records. (https://tools.ietf.org/html/rfc4180) Note that we allow leading and trailing spaces with int or float field.

Args:

- `records`: A `Tensor` of type `string`. Each string is a record/row in the csv and all records should have the same format.
- `record_defaults`: A list of `Tensor` objects with types

  from: `float32`, `int32`, `int64`, `string`. One tensor per column of the input record, with either a scalar default value for that column or empty if the column is required.
- `field_delim`: An optional `string`. Defaults to `","`. delimiter to separate fields in a record.
- `name`: A name for the operation (optional).

Returns:

A list of `Tensor` objects. Has the same type as `record_defaults`. Each tensor will have the same shape as records.

```
tf.decode_raw(bytes, out_type, little_endian=None, name=None)
```

Reinterpret the bytes of a string as a vector of numbers.

Args:

- `bytes`: A `Tensor` of type `string`. All the elements must have the same length.
- `out_type`: A `tf.DType` from: `tf.float32, tf.float64, tf.int32, tf.uint8, tf.int16, tf.int8, tf.int64`.
- `little_endian`: An optional `bool`. Defaults to `True`. Whether the input `bytes` are in little-endian order. Ignored for `out_type` values that are stored in a single byte like `uint8`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `out_type`. A Tensor with one more dimension than the input `bytes`. The added dimension will have size equal to the length of the elements of `bytes` divided by the number of bytes to represent`out_type`.

Example protocol buffer

TensorFlow's recommended format for training examples is serialized `Example` protocol buffers, described here. They contain `Features`, described here.

`class tf.VarLenFeature`

Configuration for parsing a variable-length input feature.

Fields: dtype: Data type of input.

`tf.VarLenFeature.dtype`

Alias for field number 0

`class tf.FixedLenFeature`

Configuration for parsing a fixed-length input feature.

To treat sparse input as dense, provide a `default_value`; otherwise, the parse functions will fail on any examples missing this feature. Fields: shape: Shape of input data. dtype: Data type of input. default_value: Value to be used if an example is missing this feature. It must be compatible with `dtype`.

`tf.FixedLenFeature.default_value`

Alias for field number 2

`tf.FixedLenFeature.dtype`

Alias for field number 1

`tf.FixedLenFeature.shape`

Alias for field number 0

`class tf.FixedLenSequenceFeature`

Configuration for a dense input feature in a sequence item.

To treat a sparse input as dense, provide `allow_missing=True`; otherwise, the parse functions will fail on any examples missing this feature.

Fields: shape: Shape of input data. dtype: Data type of input. allow_missing: Whether to allow this feature to be missing from a feature list item.

`tf.FixedLenSequenceFeature.allow_missing`

Alias for field number 2

`tf.FixedLenSequenceFeature.dtype`

Alias for field number 1

`tf.FixedLenSequenceFeature.shape`

Alias for field number 0

```
tf.parse_example(serialized, features, name=None,
example_names=None)
```

Parses `Example` protos into a `dict` of tensors.

Parses a number of serialized `Example` protos given in `serialized`.

`example_names` may contain descriptive names for the corresponding serialized protos. These may be useful for debugging purposes, but they have no effect on the output. If not `None`, `example_names` must be the same length as `serialized`.

This op parses serialized examples into a dictionary mapping keys to `Tensor` and `SparseTensor` objects.`features` is a dict from keys to `VarLenFeature` and `FixedLenFeature` objects.

Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenFeature` is mapped to a `Tensor`.

Each `VarLenFeature` maps to a `SparseTensor` of the specified type representing a ragged matrix. Its indices are `[batch, index]` where `batch` is the batch entry the value is from in `serialized`, and `index` is the value's index in the list of values associated with that feature and example.

Each `FixedLenFeature` `df` maps to a `Tensor` of the specified type (or `tf.float32` if not specified) and shape `(serialized.size(),) + df.shape`.

`FixedLenFeature` entries with a `default_value` are optional. With no default value, we will fail if that`Feature` is missing from any example in `serialized`.

Examples:

For example, if one expects a `tf.float32` sparse feature `ft` and three serialized `Example`s are provided:

```
serialized = [
  features
    { feature { key: "ft" value { float_list { value: [1.0,
2.0] } } } },
  features
    { feature []},
  features
    { feature { key: "ft" value { float_list { value: [3.0] } } }
]
```

then the output will look like:

```
{"ft": SparseTensor(indices=[[0, 0], [0, 1], [2, 0]],
                    values=[1.0, 2.0, 3.0],
                    shape=(3, 2)) }
```

Given two `Example` input protos in `serialized`:

```
[
  features {
    feature { key: "kw" value { bytes_list { value: [ "knit",
"big" ] } } }
    feature { key: "gps" value { float_list { value: [] } } }
  },
  features {
    feature { key: "kw" value { bytes_list { value:
[ "emmy" ] } } }
    feature { key: "dank" value { int64_list { value: [ 42 ] } } }
    feature { key: "gps" value { } }
  }
]
```

And arguments

```
example_names: ["input0", "input1"],
features: {
    "kw": VarLenFeature(tf.string),
    "dank": VarLenFeature(tf.int64),
    "gps": VarLenFeature(tf.float),
}
```

Then the output is a dictionary:

```
{
  "kw": SparseTensor(
      indices=[[0, 0], [0, 1], [1, 0]],
      values=["knit", "big", "emmy"]
      shape=[2, 2]),
  "dank": SparseTensor(
      indices=[[1, 0]],
      values=[42],
      shape=[2, 1]),
  "gps": SparseTensor(
      indices=[],
      values=[],
      shape=[2, 0]),
}
```

For dense results in two serialized `Example`s:

```
[
  features {
    feature { key: "age" value { int64_list { value: [ 0 ] } } }
```

```
    feature { key: "gender" value { bytes_list { value:
[ "f" ] } } }
   },
   features {
    feature { key: "age" value { int64_list { value: [] } } }
    feature { key: "gender" value { bytes_list { value:
[ "f" ] } } }
  }
]
```

We can use arguments:

```
example_names: ["input0", "input1"],
features: {
    "age": FixedLenFeature([], dtype=tf.int64, default_value=-1),
    "gender": FixedLenFeature([], dtype=tf.string),
}
```

And the expected output is:

```
{
  "age": [[0], [-1]],
  "gender": [["f"], ["f"]],
}
```

Args:

- `serialized`: A vector (1-D Tensor) of strings, a batch of binary serialized `Example` protos.
- `features`: A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature` values.
- `name`: A name for this operation (optional).
- `example_names`: A vector (1-D Tensor) of strings (optional), the names of the serialized protos in the batch.

Returns:

A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:

- `ValueError`: if any feature is invalid.

`tf.parse_single_example(serialized, features, name=None, example_names=None)`

Parses a single `Example` proto.

Similar to `parse_example`, except:

For dense tensors, the returned `Tensor` is identical to the output of `parse_example`, except there is no batch dimension, the output shape is the same as the shape given in `dense_shape`.

For `SparseTensor`s, the first (batch) column of the indices matrix is removed (the indices matrix is a column vector), the values vector is unchanged, and the first (`batch_size`) entry of the shape vector is removed (it is now a single element vector).

Args:

- `serialized`: A scalar string Tensor, a single serialized Example. See `_parse_single_example_raw`documentation for more details.
- `features`: A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature` values.
- `name`: A name for this operation (optional).

- `example_names`: (Optional) A scalar string Tensor, the associated name. See `_parse_single_example_raw` documentation for more details.

Returns:

A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:

- `ValueError`: if any feature is invalid.

`tf.decode_json_example(json_examples, name=None)`

Convert JSON-encoded Example records to binary protocol buffer strings.

This op translates a tensor containing Example records, encoded using the [standard JSON mapping](), into a tensor containing the same records encoded as binary protocol buffers. The resulting tensor can then be fed to any of the other Example-parsing ops.

Args:

- `json_examples`: A `Tensor` of type `string`. Each string is a JSON object serialized according to the JSON mapping of the Example proto.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `string`. Each string is a binary Example protocol buffer corresponding to the respective element of `json_examples`.

Queues

TensorFlow provides several implementations of 'Queues', which are structures within the TensorFlow computation graph to stage pipelines of tensors together. The following describe the basic Queue interface and some implementations. To see an example use, see Threading and Queues.

`class tf.QueueBase`

Base class for queue implementations.

A queue is a TensorFlow data structure that stores tensors across multiple steps, and exposes operations that enqueue and dequeue tensors.

Each queue element is a tuple of one or more tensors, where each tuple component has a static dtype, and may have a static shape. The queue implementations support versions of enqueue and dequeue that handle single elements, versions that support enqueuing and dequeuing a batch of elements at once.

See `tf.FIFOQueue` and `tf.RandomShuffleQueue` for concrete implementations of this class, and instructions on how to create them.

`tf.QueueBase.enqueue(vals, name=None)`

Enqueues one element to this queue.

If the queue is full when this operation executes, it will block until the element has been enqueued.

Args:

- `vals`: The tuple of `Tensor` objects to be enqueued.
- `name`: A name for the operation (optional).

Returns:

The operation that enqueues a new tuple of tensors to the queue.

`tf.QueueBase.enqueue_many(vals, name=None)`

Enqueues zero or elements to this queue.

This operation slices each component tensor along the 0th dimension to make multiple queue elements. All of the tensors in `vals` must have the same size in the 0th dimension.

If the queue is full when this operation executes, it will block until all of the elements have been enqueued.

Args:

- `vals`: The tensor or tuple of tensors from which the queue elements are taken.
- `name`: A name for the operation (optional).

Returns:

The operation that enqueues a batch of tuples of tensors to the queue.

`tf.QueueBase.dequeue(name=None)`

Dequeues one element from this queue.

If the queue is empty when this operation executes, it will block until there is an element to dequeue.

Args:

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

Returns:

The tuple of tensors that was dequeued.

`tf.QueueBase.dequeue_many(n, name=None)`

Dequeues and concatenates `n` elements from this queue.

This operation concatenates queue-element component tensors along the 0th dimension to make a single component tensor. All of the components in the dequeued tuple will have size `n` in the 0th dimension.

If the queue contains fewer than `n` elements when this operation executes, it will block until `n` elements have been dequeued.

Args:

- `n`: A scalar `Tensor` containing the number of elements to dequeue.
- `name`: A name for the operation (optional).

Returns:

The tuple of concatenated tensors that was dequeued.

`tf.QueueBase.size(name=None)`

Compute the number of elements in this queue.

Args:

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

Returns:

A scalar tensor containing the number of elements in this queue.

`tf.QueueBase.close(cancel_pending_enqueues=False, name=None)`

Closes this queue.

This operation signals that no more elements will be enqueued in the given queue. Subsequent `enqueue` and`enqueue_many` operations will fail. Subsequent `dequeue` and `dequeue_many` operations will continue to succeed if sufficient elements remain in the queue.

Subsequent `dequeue` and `dequeue_many` operations that would block will fail immediately.
If `cancel_pending_enqueues` is `True`, all pending requests will also be cancelled.

Args:

- `cancel_pending_enqueues`: (Optional.) A boolean, defaulting to `False` (described above).
- `name`: A name for the operation (optional).

Returns:

The operation that closes the queue.

Other Methods

`tf.QueueBase.__init__(dtypes, shapes, queue_ref)`

Constructs a queue object from a queue reference.

Args:

- `dtypes`: A list of types. The length of dtypes must equal the number of tensors in each element.
- `shapes`: Constraints on the shapes of tensors in an element: A list of shape tuples or None. This list is the same length as dtypes. If the shape of any tensors in the element are constrained, all must be; shapes can be None if the shapes should not be constrained.

- `queue_ref`: The queue reference, i.e. the output of the queue op.

`tf.QueueBase.dtypes`

The list of dtypes for each component of a queue element.

`tf.QueueBase.from_list(index, queues)`

Create a queue using the queue reference from `queues[index]`.

Args:

- `index`: An integer scalar tensor that determines the input that gets selected.
- `queues`: A list of `QueueBase` objects.

Returns:

A `QueueBase` object.

Raises:

- `TypeError`: When `queues` is not a list of `QueueBase` objects, or when the data types of `queues` are not all the same.

`tf.QueueBase.name`

The name of the underlying queue.

`tf.QueueBase.queue_ref`

The underlying queue reference.

`class tf.FIFOQueue`

A queue implementation that dequeues elements in first-in-first out order.

See `tf.QueueBase` for a description of the methods on this class.

`tf.FIFOQueue.__init__(capacity, dtypes, shapes=None, shared_name=None, name='fifo_queue')`

Creates a queue that dequeues elements in a first-in first-out order.

A `FIFOQueue` has bounded capacity; supports multiple concurrent producers and consumers; and provides exactly-once delivery.
A `FIFOQueue` holds a list of up to `capacity` elements. Each element is a fixed-length tuple of tensors whose dtypes are described by `dtypes`, and whose shapes are optionally described by

the `shapes` argument.

If the `shapes` argument is specified, each component of a queue element must have the respective fixed shape. If it is unspecified, different queue elements may have different shapes, but the use of `dequeue_many` is disallowed.

Args:

- `capacity`: An integer. The upper bound on the number of elements that may be stored in this queue.
- `dtypes`: A list of `DType` objects. The length of `dtypes` must equal the number of tensors in each queue element.
- `shapes`: (Optional.) A list of fully-defined `TensorShape` objects, with the same length as `dtypes` or `None`.
- `shared_name`: (Optional.) If non-empty, this queue will be shared under the given name across multiple sessions.
- `name`: Optional name for the queue operation.

`class tf.RandomShuffleQueue`

A queue implementation that dequeues elements in a random order.

See `tf.QueueBase` for a description of the methods on this class.

```
tf.RandomShuffleQueue.__init__(capacity,
min_after_dequeue, dtypes, shapes=None, seed=None,
shared_name=None, name='random_shuffle_queue')
```

Create a queue that dequeues elements in a random order.

A `RandomShuffleQueue` has bounded capacity; supports multiple concurrent producers and consumers; and provides exactly-once delivery.

A `RandomShuffleQueue` holds a list of up to `capacity` elements. Each element is a fixed-length tuple of tensors whose dtypes are described by `dtypes`, and whose shapes are optionally described by the `shapes`argument.

If the `shapes` argument is specified, each component of a queue element must have the respective fixed shape. If it is unspecified, different queue elements may have different shapes, but the use of `dequeue_many` is disallowed.

The `min_after_dequeue` argument allows the caller to specify a minimum number of elements that will remain in the queue after a `dequeue` or `dequeue_many` operation completes, to ensure a minimum level of mixing of elements. This invariant is maintained by blocking those operations until sufficient elements have been enqueued. The `min_after_dequeue` argument is ignored after the queue has been closed.

Args:

- `capacity`: An integer. The upper bound on the number of elements that may be stored in this queue.
- `min_after_dequeue`: An integer (described above).
- `dtypes`: A list of `DType` objects. The length of `dtypes` must equal the number of tensors in each queue element.
- `shapes`: (Optional.) A list of fully-defined `TensorShape` objects, with the same length as `dtypes` or `None`.

* `seed`: A Python integer. Used to create a random seed.

  See `set_random_seed` for behavior.

* `shared_name`: (Optional.) If non-empty, this queue will be shared under the given name across multiple sessions.
* `name`: Optional name for the queue operation.

Dealing with the filesystem

`tf.matching_files(pattern, name=None)`

Returns the set of files matching a pattern.

Note that this routine only supports wildcard characters in the basename portion of the pattern, not in the directory portion.

Args:

* `pattern`: A `Tensor` of type `string`. A (scalar) shell wildcard pattern.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `string`. A vector of matching filenames.

`tf.read_file(filename, name=None)`

Reads and outputs the entire contents of the input filename.

Args:

- `filename`: A `Tensor` of type `string`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

Input pipeline

TensorFlow functions for setting up an input-prefetching pipeline. Please see the reading data how-to for context.

Beginning of an input pipeline

The "producer" functions add a queue to the graph and a corresponding `QueueRunner` for running the subgraph that fills that queue.

`tf.train.match_filenames_once(pattern, name=None)`

Save the list of files matching pattern, so it is only computed once.

Args:

- `pattern`: A file pattern (glob).

- `name`: A name for the operations (optional).

Returns:

A variable that is initialized to the list of files matching pattern.

`tf.train.limit_epochs(tensor, num_epochs=None, name=None)`

Returns tensor `num_epochs` times and then raises an `OutOfRange` error.

Args:

- `tensor`: Any `Tensor`.
- `num_epochs`: A positive integer (optional). If specified, limits the number of steps the output tensor may be evaluated.
- `name`: A name for the operations (optional).

Returns:

tensor or `OutOfRange`.

Raises:

- `ValueError`: if `num_epochs` is invalid.

`tf.train.range_input_producer(limit, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`

Produces the integers from 0 to limit-1 in a queue.

Args:

- `limit`: An int32 scalar tensor.
- `num_epochs`: An integer (optional). If specified, `range_input_producer` produces each integer`num_epochs` times before generating an OutOfRange error. If not specified, `range_input_producer` can cycle through the integers an unlimited number of times.
- `shuffle`: Boolean. If true, the integers are randomly shuffled within each epoch.
- `seed`: An integer (optional). Seed used if shuffle == True.
- `capacity`: An integer. Sets the queue capacity.
- `name`: A name for the operations (optional).

Returns:

A Queue with the output integers. A `QueueRunner` for the Queue is added to the current `Graph`'s`QUEUE_RUNNER` collection.

`tf.train.slice_input_producer(tensor_list, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`

Produces a slice of each `Tensor` in `tensor_list`.

Implemented using a Queue -- a `QueueRunner` for the Queue is added to the current `Graph`'s `QUEUE_RUNNER`collection.

Args:

- `tensor_list`: A list of `Tensor` objects.

  Every `Tensor` in `tensor_list` must have the same size in the first dimension.
- `num_epochs`: An integer (optional). If specified, `slice_input_producer` produces each slice`num_epochs` times before generating an `OutOfRange` error. If not specified, `slice_input_producer`can cycle through the slices an unlimited number of times.
- `shuffle`: Boolean. If true, the integers are randomly shuffled within each epoch.
- `seed`: An integer (optional). Seed used if shuffle == True.
- `capacity`: An integer. Sets the queue capacity.
- `name`: A name for the operations (optional).

Returns:

A list of tensors, one for each element of `tensor_list`. If the tensor in `tensor_list` has shape `[N, a, b, .., z]`, then the corresponding output tensor will have shape `[a, b, ..., z]`.

Raises:

- `ValueError`: if `slice_input_producer` produces nothing from `tensor_list`.

`tf.train.string_input_producer(string_tensor, num_epochs=None, shuffle=True, seed=None, capacity=32, name=None)`

Output strings (e.g. filenames) to a queue for an input pipeline.

Args:

- `string_tensor`: A 1-D string tensor with the strings to produce.
- `num_epochs`: An integer (optional). If specified, `string_input_producer` produces each string from`string_tensor` `num_epochs` times before generating an OutOfRange error. If not specified,`string_input_producer` can cycle through the strings in `string_tensor` an unlimited number of times.
- `shuffle`: Boolean. If true, the strings are randomly shuffled within each epoch.
- `seed`: An integer (optional). Seed used if shuffle == True.
- `capacity`: An integer. Sets the queue capacity.
- `name`: A name for the operations (optional).

Returns:

A queue with the output strings. A `QueueRunner` for the Queue is added to the current `Graph`'s `QUEUE_RUNNER`collection.

Raises:

- `ValueError`: If the string_tensor is a null Python list. At runtime, will fail with an assertion if string_tensor becomes a null tensor.

Batching at the end of an input pipeline

These functions add a queue to the graph to assemble a batch of examples, with possible shuffling. They also add a `QueueRunner` for running the subgraph that fills that queue.

Use `batch` or `batch_join` for batching examples that have already been well shuffled. Use `shuffle_batch`or `shuffle_batch_join` for examples that would benefit from additional shuffling.

Use `batch` or `shuffle_batch` if you want a single thread producing examples to batch, or if you have a single subgraph producing examples but you want to run it in N threads (where you increase N until it can keep the queue full).

Use `batch_join` or `shuffle_batch_join` if you have N different subgraphs producing examples to batch and you want them run by N threads.

```
tf.train.batch(tensor_list, batch_size, num_threads=1,
capacity=32, enqueue_many=False, shapes=None, name=None)
```

Creates batches of tensors in `tensor_list`.

This function is implemented using a queue. A `QueueRunner` for the queue is added to the current `Graph`'s`QUEUE_RUNNER` collection.

If `enqueue_many` is `False`, `tensor_list` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensor_list` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.
The returned operation is a dequeue operation and will throw `tf.errors.OutOfRangeError` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

N.B.: You must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensor_list`must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

Args:

- `tensor_list`: The list of tensors to enqueue.
- `batch_size`: The new batch size pulled from the queue.
- `num_threads`: The number of threads enqueuing `tensor_list`.

- `capacity`: An integer. The maximum number of elements in the queue.
- `enqueue_many`: Whether each tensor in `tensor_list` is a single example.
- `shapes`: (Optional) The shapes for each example. Defaults to the inferred shapes for `tensor_list`.
- `name`: (Optional) A name for the operations.

Returns:

A list of tensors with the same number and types as `tensor_list`.

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of `tensor_list`.

```
tf.train.batch_join(tensor_list_list, batch_size,
capacity=32, enqueue_many=False, shapes=None, name=None)
```

Runs a list of tensors to fill a queue to create batches of examples.

Enqueues a different list of tensors in different threads. Implemented using a queue -- a `QueueRunner` for the queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

`len(tensor_list_list)` threads will be started, with thread `i` enqueuing the tensors from`tensor_list_list[i]`. `tensor_list_list[i1][j]` must match `tensor_list_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensor_list_list[i]` is assumed to represent a single example. An input tensor `x` will be output as a tensor with shape `[batch_size] + x.shape`.

If `enqueue_many` is `True`, `tensor_list_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list_list[i]` should have the same size in the first dimension. The slices of any input tensor `x` are treated as examples, and the output tensors will have shape `[batch_size] + x.shape[1:]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.
The returned operation is a dequeue operation and will throw `tf.errors.OutOfRangeError` if the input queue is exhausted.
If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

N.B.: You must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in`tensor_list_list` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

Args:

- `tensor_list_list`: A list of tuples of tensors to enqueue.

- `batch_size`: An integer. The new batch size pulled from the queue.
- `capacity`: An integer. The maximum number of elements in the queue.
- `enqueue_many`: Whether each tensor in `tensor_list_list` is a single example.
- `shapes`: (Optional) The shapes for each example. Defaults to the inferred shapes for`tensor_list_list[i]`.
- `name`: (Optional) A name for the operations.

Returns:

A list of tensors with the same number and types as `tensor_list_list[i]`.

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of`tensor_list_list`.

```
tf.train.shuffle_batch(tensor_list, batch_size, capacity,
min_after_dequeue, num_threads=1, seed=None,
enqueue_many=False, shapes=None, name=None)
```

Creates batches by randomly shuffling tensors.

This function adds the following to the current `Graph`:

- A shuffling queue into which tensors from `tensor_list` are enqueued.

- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensor_list`.

If `enqueue_many` is `False`, `tensor_list` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensor_list` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.
The returned operation is a dequeue operation and will throw `tf.errors.OutOfRangeError` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

For example:

```
# Creates batches of 32 images and 32 labels.
image_batch, label_batch = tf.train.shuffle_batch(
     [single_image, single_label],
     batch_size=32,
     num_threads=4,
     capacity=50000,
     min_after_dequeue=10000)
```

N.B.: You must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensor_list`must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

Args:

- `tensor_list`: The list of tensors to enqueue.
- `batch_size`: The new batch size pulled from the queue.
- `capacity`: An integer. The maximum number of elements in the queue.
- `min_after_dequeue`: Minimum number elements in the queue after a dequeue, used to ensure a level of mixing of elements.
- `num_threads`: The number of threads enqueuing `tensor_list`.
- `seed`: Seed for the random shuffling within the queue.
- `enqueue_many`: Whether each tensor in `tensor_list` is a single example.
- `shapes`: (Optional) The shapes for each example. Defaults to the inferred shapes for `tensor_list`.
- `name`: (Optional) A name for the operations.

Returns:

A list of tensors with the same number and types as `tensor_list`.

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of `tensor_list`.

`tf.train.shuffle_batch_join(tensor_list_list, batch_size, capacity, min_after_dequeue, seed=None, enqueue_many=False, shapes=None, name=None)`

Create batches by randomly shuffling tensors.

This version enqueues a different list of tensors in different threads. It adds the following to the current `Graph`:

- A shuffling queue into which tensors from `tensor_list_list` are enqueued.
- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensor_list_list`.

`len(tensor_list_list)` threads will be started, with thread `i` enqueuing the tensors from`tensor_list_list[i]`. `tensor_list_list[i1][j]` must match `tensor_list_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensor_list_list[i]` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensor_list_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list_list[i]` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.
The returned operation is a dequeue operation and will throw `tf.errors.OutOfRangeError` if the input queue is exhausted.

If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

Args:

- `tensor_list_list`: A list of tuples of tensors to enqueue.
- `batch_size`: An integer. The new batch size pulled from the queue.
- `capacity`: An integer. The maximum number of elements in the queue.
- `min_after_dequeue`: Minimum number elements in the queue after a dequeue, used to ensure a level of mixing of elements.
- `seed`: Seed for the random shuffling within the queue.
- `enqueue_many`: Whether each tensor in `tensor_list_list` is a single example.
- `shapes`: (Optional) The shapes for each example. Defaults to the inferred shapes for`tensor_list_list[i]`.
- `name`: (Optional) A name for the operations.

Returns:

A list of tensors with the same number and types as `tensor_list_list[i]`.

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of`tensor_list_list`.

Data IO (Python functions)

Contents

Data IO (Python Functions)

A TFRecords file represents a sequence of (binary) strings. The format is not random access, so it is suitable for streaming large amounts of data but not suitable if fast sharding or other non-sequential access is desired.

`class tf.python_io.TFRecordWriter`

A class to write records to a TFRecords file.

This class implements `__enter__` and `__exit__`, and can be used in `with` blocks like a normal file.

`tf.python_io.TFRecordWriter.__init__(path)`

Opens file `path` and creates a `TFRecordWriter` writing to it.

Args:

- `path`: The path to the TFRecords file.

Raises:

- `IOError`: If `path` cannot be opened for writing.

`tf.python_io.TFRecordWriter.write(record)`

Write a string record to the file.

Args:

- `record`: str

`tf.python_io.TFRecordWriter.close()`

Close the file.

`tf.python_io.tf_record_iterator(path)`

An iterator that read the records from a TFRecords file.

Args:

- `path`: The path to the TFRecords file.

Yields:

Strings.

Raises:

- `IOError`: If `path` cannot be opened for reading.

TFRecords Format Details

A TFRecords file contains a sequence of strings with CRC hashes. Each record has the format

```
uint64 length
uint32 masked_crc32_of_length
byte   data[length]
uint32 masked_crc32_of_data
```

and the records are concatenated together to produce the file. The CRC32s are [described here](), and the mask of a CRC is

```
masked_crc = ((crc >> 15) | (crc << 17)) + 0xa282ead8ul
```

Neural Network

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

Contents

- `tf.nn.relu(features, name=None)`
- `tf.nn.relu6(features, name=None)`
- `tf.nn.elu(features, name=None)`
- `tf.nn.softplus(features, name=None)`
- `tf.nn.softsign(features, name=None)`
- `tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None, name=None)`
- `tf.nn.bias_add(value, bias, name=None)`
- `tf.sigmoid(x, name=None)`
- `tf.tanh(x, name=None)`
- Convolution
- `tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, name=None)`
- `tf.nn.depthwise_conv2d(input, filter, strides, padding, name=None)`
- `tf.nn.separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding, name=None)`
- `tf.nn.conv2d_transpose(value, filter, output_shape, strides, padding=SAME, name=None)`
- Pooling
- `tf.nn.avg_pool(value, ksize, strides, padding, name=None)`
- `tf.nn.max_pool(value, ksize, strides, padding, name=None)`
- `tf.nn.max_pool_with_argmax(input, ksize, strides, padding, Targmax=None, name=None)`
- Normalization
- `tf.nn.l2_normalize(x, dim, epsilon=1e-12, name=None)`
- `tf.nn.local_response_normalization(input, depth_radius=None, bias=None, alpha=None, beta=None, name=None)`
- `tf.nn.moments(x, axes, name=None, keep_dims=False)`
- Losses
- `tf.nn.l2_loss(t, name=None)`
- Classification
- `tf.nn.sigmoid_cross_entropy_with_logits(logits, targets, name=None)`
- `tf.nn.softmax(logits, name=None)`
- `tf.nn.softmax_cross_entropy_with_logits(logits, labels, name=None)`
- `tf.nn.sparse_softmax_cross_entropy_with_logits(logits, labels, name=None)`
- Embeddings
- `tf.nn.embedding_lookup(params, ids, partition_strategy=mod, name=None, validate_indices=True)`
- Evaluation

- `tf.nn.top_k(input, k=1, sorted=True, name=None)`
- `tf.nn.in_top_k(predictions, targets, k, name=None)`
- Candidate Sampling
- Sampled Loss Functions
- `tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=False, partition_strategy=mod, name=nce_loss)`
- `tf.nn.sampled_softmax_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=True, partition_strategy=mod, name=sampled_softmax_loss)`
- Candidate Samplers
- `tf.nn.uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
- `tf.nn.log_uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
- `tf.nn.learned_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
- `tf.nn.fixed_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, vocab_file=, distortion=1.0, num_reserved_ids=0, num_shards=1, shard=0, unigrams=(), seed=None, name=None)`
- Miscellaneous candidate sampling utilities
- `tf.nn.compute_accidental_hits(true_classes, sampled_candidates, num_true, seed=None, name=None)`

Activation Functions

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities (`sigmoid`, `tanh`, `elu`, `softplus`, and `softsign`), continuous but not everywhere differentiable functions (`relu`, `relu6`, and `relu_x`), and random regularization (`dropout`).

All activation ops apply componentwise, and produce a tensor of the same shape as the input tensor.

`tf.nn.relu(features, name=None)`

Computes rectified linear: `max(features, 0)`.

Args:

- `features`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`,`int16`, `int8`, `uint16`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.nn.relu6(features, name=None)`

Computes Rectified Linear 6: `min(max(features, 0), 6)`.

Args:

- `features`: A `Tensor` with type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` with the same type as `features`.

`tf.nn.elu(features, name=None)`

Computes exponential linear: `exp(features) - 1` if <

0, `features` otherwise.

See Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

Args:

- `features`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.nn.softplus(features, name=None)`

Computes softplus: `log(exp(features) + 1)`.

Args:

* `features`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`,`int16`, `int8`, `uint16`.
* `name`: A name for the operation (optional).

Returns:

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

`tf.nn.softsign(features, name=None)`

Computes softsign: `features / (abs(features) + 1)`.

Args:

* `features`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`,`int16`, `int8`, `uint16`.
* `name`: A name for the operation (optional).

Returns:

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

```
tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None,
name=None)
```

Computes dropout.

With probability `keep_prob`, outputs the input element scaled up by `1 / keep_prob`, otherwise outputs `0`. The scaling is so that the expected sum is unchanged.
By default, each element is kept or dropped independently.
If `noise_shape` is specified, it must be broadcastableto the shape of `x`, and only dimensions with `noise_shape[i] == shape(x)[i]` will make independent decisions. For example, if `shape(x) = [k, l, m, n]` and `noise_shape = [k, 1, 1, n]`, each batch and channel component will be kept independently and each row and column will be kept or not kept together.

Args:

- `x`: A tensor.
- `keep_prob`: A scalar `Tensor` with the same type as x. The probability that each element is kept.
- `noise_shape`: A 1-D `Tensor` of type `int32`, representing the shape for randomly generated keep/drop flags.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `name`: A name for this operation (optional).

Returns:

A Tensor of the same shape of `x`.

Raises:

* `ValueError`: If `keep_prob` is not in `(0, 1]`.

`tf.nn.bias_add(value, bias, name=None)`

Adds `bias` to `value`.

This is (mostly) a special case of `tf.add` where `bias` is restricted to 1-D. Broadcasting is supported, so `value`may have any number of dimensions. Unlike `tf.add`, the type of `bias` is allowed to differ from `value` in the case where both types are quantized.

Args:

* `value`: A `Tensor` with type `float`, `double`, `int64`, `int32`, `uint8`, `int16`, `int8`, or `complex64`.
* `bias`: A 1-D `Tensor` with size matching the last dimension of `value`. Must be the same type as `value`unless `value` is a quantized type, in which case a different quantized type may be used.
* `name`: A name for the operation (optional).

Returns:

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

`tf.sigmoid(x, name=None)`

Computes sigmoid of `x` element-wise.

Specifically, `y = 1 / (1 + exp(-x))`.

Args:

- `x`: A Tensor with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

`tf.tanh(x, name=None)`

Computes hyperbolic tangent of `x` element-wise.

Args:

- `x`: A Tensor with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

Convolution

The convolution ops sweep a 2-D filter over a batch of images, applying the filter to each window of each image of the appropriate size. The different ops trade off between generic vs. specific filters:

- `conv2d`: Arbitrary filters that can mix channels together.
- `depthwise_conv2d`: Filters that operate on each channel independently.
- `separable_conv2d`: A depthwise spatial filter followed by a pointwise filter.

Note that although these ops are called "convolution", they are strictly speaking "cross-correlation" since the filter is combined with an input window without reversing the filter. For details, see [the properties of cross-correlation](#).

The filter is applied to image patches of the same size as the filter and strided according to the `strides`argument. `strides = [1, 1, 1, 1]` applies the filter to a patch at every offset, `strides = [1, 2, 2, 1]` applies the filter to every other image patch in each dimension, etc.

Ignoring channels for the moment, and assume that the 4-D `input` has shape `[batch, in_height, in_width, ...]` and the 4-D `filter` has shape `[filter_height, filter_width, ...]`, then the spatial semantics of the convolution ops are as follows: first, according to the padding scheme chosen as `'SAME'`or `'VALID'`, the output size and the padding pixels are computed. For the `'SAME'` padding, the output height and width are computed as:

```
out_height = ceil(float(in_height) / float(strides[1]))
```

```
out_width  = ceil(float(in_width) / float(strides[2]))
```

and the padding on the top and left are computed as:

```
pad_along_height = ((out_height - 1) * strides[1] +
                  filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                 filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2
```

Note that the division by 2 means that there might be cases when the padding on both sides (top vs bottom, right vs left) are off by one. In this case, the bottom and right sides always get the one additional padded pixel. For example, when `pad_along_height` is 5, we pad 2 pixels at the top and 3 pixels at the bottom. Note that this is different from existing libraries such as cuDNN and Caffe, which explicitly specify the number of padded pixels and always pad the same number of pixels on both sides.

For the `'VALID`' padding, the output height and width are computed as:

```
out_height = ceil(float(in_height - filter_height + 1) /
float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) /
float(strides[2]))
```

and the padding values are always zero. The output is then computed as

```
output[b, i, j, :] =
    sum_{di, dj} input[b, strides[1] * i + di - pad_top,
                   strides[2] * j + dj - pad_left, ...] *
               filter[di, dj, ...]
```

where any value outside the original input image region are considered zero (i.e. we pad zero values around the border of the image).

Since `input` is 4-D, each `input[b, i, j, :]` is a vector.

For `conv2d`, these vectors are multiplied by the`filter[di, dj, :, :]` matrices to produce new vectors.

For `depthwise_conv_2d`, each scalar component `input[b, i, j, k]` is multiplied by a vector `filter[di, dj, k]`, and all the vectors are concatenated.

`tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, name=None)`

Computes a 2-D convolution given 4-D `input` and `filter` tensors.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a virtual tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

In detail,

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j +
dj, q] *
                    filter[di, dj, q, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertices strides, `strides = [1, stride, stride, 1]`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `filter`: A `Tensor`. Must have the same type as `input`.
- `strides`: A list of `ints`. 1-D of length 4. The stride of the sliding window for each dimension of `input`.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `use_cudnn_on_gpu`: An optional `bool`. Defaults to `True`.
- `name`: A name for the operation (optional).

Returns:

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

```
tf.nn.depthwise_conv2d(input, filter, strides, padding, name=None)
```

Depthwise 2-D convolution.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter tensor of shape `[filter_height, filter_width, in_channels, channel_multiplier]` containing`in_channels` convolutional filters of depth 1, `depthwise_conv2d` applies a different filter to each input channel (expanding from 1 channel to `channel_multiplier` channels for each), then concatenates the results together. The output has `in_channels * channel_multiplier` channels.

In detail,

```
output[b, i, j, k * channel_multiplier + q] =
    sum_{di, dj} input[b, strides[1] * i + di, strides[2] * j + dj, k] *
                 filter[di, dj, k, q]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`.

Args:

- `input`: 4-D with shape `[batch, in_height, in_width, in_channels]`.
- `filter`: 4-D with shape `[filter_height, filter_width, in_channels, channel_multiplier]`.
- `strides`: 1-D of size 4. The stride of the sliding window for each dimension of `input`.
- `padding`: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- `name`: A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape `[batch, out_height, out_width, in_channels * channel_multiplier]`.

```
tf.nn.separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding, name=None)
```

2-D convolution with separable filters.

Performs a depthwise convolution that acts separately on channels followed by a pointwise convolution that mixes channels. Note that this is separability between dimensions `[1, 2]` and `3`, not spatial separability between dimensions `1` and `2`.

In detail,

```
output[b, i, j, k] = sum_{di, dj, q, r]
    input[b, strides[1] * i + di, strides[2] * j + dj, q] *
    depthwise_filter[di, dj, q, r] *
    pointwise_filter[0, 0, q * channel_multiplier + r, k]
```

`strides` controls the strides for the depthwise convolution only, since the pointwise convolution has implicit strides of `[1, 1, 1, 1]`. Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`.

Args:

- `input`: 4-D `Tensor` with shape `[batch, in_height, in_width, in_channels]`.

- `depthwise_filter`: 4-D `Tensor` with shape `[filter_height, filter_width, in_channels, channel_multiplier]`. Contains `in_channels` convolutional filters of depth 1.
- `pointwise_filter`: 4-D `Tensor` with shape `[1, 1, channel_multiplier * in_channels, out_channels]`. Pointwise filter to mix channels after `depthwise_filter` has convolved spatially.
- `strides`: 1-D of size 4. The strides for the depthwise convolution for each dimension of `input`.
- `padding`: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- `name`: A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape `[batch, out_height, out_width, out_channels]`.

```
tf.nn.conv2d_transpose(value, filter, output_shape, strides, padding='SAME', name=None)
```

The transpose of `conv2d`.

This operation is sometimes called "deconvolution" after (Deconvolutional Networks)[http://www.matthewzeiler.com/pubs/cvpr2010/cvpr2010.pdf], but is actually the transpose (gradient) of `conv2d`rather than an actual deconvolution.

Args:

- `value`: A 4-D `Tensor` of type `float` and shape `[batch, height, width, in_channels]`.
- `filter`: A 4-D `Tensor` with the same type as `value` and shape `[height, width, output_channels, in_channels]`. `filter`'s `in_channels` dimension must match that of `value`.
- `output_shape`: A 1-D `Tensor` representing the output shape of the deconvolution op.
- `strides`: A list of ints. The stride of the sliding window for each dimension of the input tensor.
- `padding`: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- `name`: Optional name for the returned tensor.

Returns:

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

Raises:

- `ValueError`: If input/output depth does not match `filter`'s shape, or if padding is other than `'VALID'` or`'SAME'`.

Pooling

The pooling ops sweep a rectangular window over the input tensor, computing a reduction operation for each window (average, max, or max with argmax). Each pooling op uses rectangular windows of size `ksize` separated by offset `strides`. For example, if `strides` is all ones every window is used, if `strides` is all twos every other window is used in each dimension, etc.

In detail, the output is

```
output[i] = reduce(value[strides * i:strides * i + ksize])
```

where the indices also take into consideration the padding values. Please refer to the `Convolution` section for details about the padding calculation.

`tf.nn.avg_pool(value, ksize, strides, padding, name=None)`

Performs the average pooling on the input.

Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.

Args:

- `value`: A 4-D `Tensor` of shape `[batch, height, width, channels]` and type `float32`, `float64`,`qint8`, `quint8`, or `qint32`.
- `ksize`: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.
- `strides`: A list of ints that has length >= 4. The stride of the sliding window for each dimension of the input tensor.
- `padding`: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- `name`: Optional name for the operation.

Returns:

A `Tensor` with the same type as `value`. The average pooled output tensor.

```
tf.nn.max_pool(value, ksize, strides, padding, name=None)
```

Performs the max pooling on the input.

Args:

- `value`: A 4-D `Tensor` with shape `[batch, height, width, channels]` and type `tf.float32`.
- `ksize`: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.
- `strides`: A list of ints that has length >= 4. The stride of the sliding window for each dimension of the input tensor.
- `padding`: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- `name`: Optional name for the operation.

Returns:

A `Tensor` with type `tf.float32`. The max pooled output tensor.

```
tf.nn.max_pool_with_argmax(input, ksize, strides,
padding, Targmax=None, name=None)
```

Performs max pooling on the input and outputs both max values and indices.

The indices in `argmax` are flattened, so that a maximum value at position `[b, y, x, c]` becomes flattened index `((b * height + y) * width + x) * channels + c`.

Args:

- `input`: A `Tensor` of type `float32`. 4-D with shape `[batch, height, width, channels]`. Input to pool over.
- `ksize`: A list of `ints` that has length `>= 4`. The size of the window for each dimension of the input tensor.
- `strides`: A list of `ints` that has length `>= 4`. The stride of the sliding window for each dimension of the input tensor.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `Targmax`: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (output, argmax).

- `output`: A `Tensor` of type `float32`. The max pooled output tensor.
- `argmax`: A `Tensor` of type `Targmax`. 4-D. The flattened indices of the max values chosen for each output.

Normalization

Normalization is useful to prevent neurons from saturating when inputs may have varying scale, and to aid generalization.

`tf.nn.l2_normalize(x, dim, epsilon=1e-12, name=None)`

Normalizes along dimension `dim` using an L2 norm.

For a 1-D tensor with `dim = 0`, computes

```
output = x / sqrt(max(sum(x**2), epsilon))
```

For `x` with more dimensions, independently normalizes each 1-D slice along dimension `dim`.

Args:

- `x`: A `Tensor`.
- `dim`: Dimension along which to normalize.
- `epsilon`: A lower bound value for the norm. Will use `sqrt(epsilon)` as the divisor if `norm < sqrt(epsilon)`.
- `name`: A name for this operation (optional).

Returns:

A `Tensor` with the same shape as `x`.

```
tf.nn.local_response_normalization(input,
depth_radius=None, bias=None, alpha=None, beta=None,
name=None)
```

Local Response Normalization.

The 4-D `input` tensor is treated as a 3-D array of 1-D vectors (along the last dimension), and each vector is normalized independently. Within a given vector, each component is divided by the weighted, squared sum of inputs within `depth_radius`. In detail,

```
sqr_sum[a, b, c, d] =
    sum(input[a, b, c, d - depth_radius : d + depth_radius + 1] **
2)
output = input / (bias + alpha * sqr_sum ** beta)
```

For details, see [Krizhevsky et al., ImageNet classification with deep convolutional neural networks (NIPS 2012)].

Args:

- `input`: A `Tensor` of type `float32`. 4-D.
- `depth_radius`: An optional `int`. Defaults to `5`. 0-D. Half-width of the 1-D normalization window.
- `bias`: An optional `float`. Defaults to `1`. An offset (usually positive to avoid dividing by 0).
- `alpha`: An optional `float`. Defaults to `1`. A scale factor, usually positive.
- `beta`: An optional `float`. Defaults to `0.5`. An exponent.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

`tf.nn.moments(x, axes, name=None, keep_dims=False)`

Calculate the mean and variance of `x`.

The mean and variance are calculated by aggregating the contents of `x` across `axes`. If `x` is 1-D and `axes = [0]` this is just the mean and variance of a vector.
For so-called "global normalization" needed for convolutional filters pass `axes=[0, 1, 2]` (batch, height, width). For batch normalization pass `axes=[0]` (batch).

Args:

- `x`: A `Tensor`.
- `axes`: array of ints. Axes along which to compute mean and variance.
- `keep_dims`: produce moments with the same dimensionality as the input.
- `name`: Name used to scope the operations that compute the moments.

Returns:

Two `Tensor` objects: `mean` and `variance`.

Losses

The loss ops measure error between two tensors, or between a tensor and zero. These can be used for measuring accuracy of a network in a regression task or for regularization purposes (weight decay).

`tf.nn.l2_loss(t, name=None)`

L2 Loss.

Computes half the L2 norm of a tensor without the `sqrt`:

```
output = sum(t ** 2) / 2
```

Args:

- `t`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`,`int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`. Typically 2-D, but may have any dimensions.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `t`. 0-D.

Classification

TensorFlow provides several operations that help you perform classification.

`tf.nn.sigmoid_cross_entropy_with_logits(logits, targets, name=None)`

Computes sigmoid cross entropy given `logits`.

Measures the probability error in discrete classification tasks in which each class is independent and not mutually exclusive. For instance, one could perform multilabel classification where a picture can contain both an elephant and a dog at the same time.

For brevity, let `x = logits`, `z = targets`. The logistic loss is

```
  z * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))
= z * -log(1 / (1 + exp(-x))) + (1 - z) * -log(exp(-x) / (1 +
exp(-x)))
= z * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-
x)))
= z * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x))
= (1 - z) * x + log(1 + exp(-x))
= x - x * z + log(1 + exp(-x))
```

To ensure stability and avoid overflow, the implementation uses

```
max(x, 0) - x * z + log(1 + exp(-abs(x)))
```

`logits` and `targets` must have the same type and shape.

Args:

- `logits`: A `Tensor` of type `float32` or `float64`.
- `targets`: A `Tensor` of the same type and shape as `logits`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `logits` with the componentwise logistic losses.

`tf.nn.softmax(logits, name=None)`

Computes softmax activations.

For each batch `i` and class `j` we have

```
softmax[i, j] = exp(logits[i, j]) / sum(exp(logits[i]))
```

Args:

- `logits`: A `Tensor`. Must be one of the following types: `float32`, `float64`. 2-D with shape `[batch_size, num_classes]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `logits`. Same shape as `logits`.

```
tf.nn.softmax_cross_entropy_with_logits(logits, labels,
name=None)
```

Computes softmax cross entropy between `logits` and `labels`.

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE:: While the classes are mutually exclusive, their probabilities need not be. All that is required is that each row of `labels` is a valid probability distribution. If using exclusive `labels` (wherein one and only one class is true at a time),

see `sparse_softmax_cross_entropy_with_logits`.

WARNING: This op expects unscaled logits, since it performs a `softmax` on `logits` internally for efficiency. Do not call this op with the output of `softmax`, as it will produce incorrect results.

`logits` and `labels` must have the same shape `[batch_size, num_classes]` and the same dtype (either`float32` or `float64`).

Args:

- `logits`: Unscaled log probabilities.
- `labels`: Each row `labels[i]` must be a valid probability distribution.
- `name`: A name for the operation (optional).

Returns:

A 1-D `Tensor` of length `batch_size` of the same type as `logits` with the softmax cross entropy loss.

```
tf.nn.sparse_softmax_cross_entropy_with_logits(logits,
labels, name=None)
```

Computes sparse softmax cross entropy between `logits` and `labels`.

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE:: For this operation, the probability of a given label is considered exclusive. That is, soft classes are not allowed, and the `labels` vector must provide a single specific index for the true class for each row of `logits`(each minibatch entry). For soft softmax classification with a probability distribution for each entry, see`softmax_cross_entropy_with_logits`.

WARNING: This op expects unscaled logits, since it performs a `softmax` on `logits` internally for efficiency. Do not call this op with the output of `softmax`, as it will produce incorrect results.

`logits` and must have the shape `[batch_size, num_classes]` and the dtype (either `float32` or`float64`).

`labels` must have the shape `[batch_size]` and the dtype `int64`.

Args:

- `logits`: Unscaled log probabilities.
- `labels`: Each entry `labels[i]` must be an index in `[0, num_classes)`.
- `name`: A name for the operation (optional).

Returns:

A 1-D `Tensor` of length `batch_size` of the same type as `logits` with the softmax cross entropy loss.

Embeddings

TensorFlow provides library support for looking up values in embedding tensors.

```
tf.nn.embedding_lookup(params, ids,
partition_strategy='mod', name=None,
validate_indices=True)
```

Looks up `ids` in a list of embedding tensors.

This function is used to perform parallel lookups on the list of tensors in `params`. It is a generalization of `tf.gather()`, where `params` is interpreted as a partition of a larger embedding tensor.

If `len(params) > 1`, each element `id` of `ids` is partitioned between the elements of `params` according to the `partition_strategy`. In all strategies, if the id space does not evenly divide the number of partitions, each of the first `(max_id + 1) % len(params)` partitions will be assigned one more id.

If `partition_strategy` is `"mod"`, we assign each id to partition `p = id % len(params)`. For instance, 13 ids are split across 5 partitions as: `[[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8], [4, 9]]`

If `partition_strategy` is `"div"`, we assign ids to partitions in a contiguous manner. In this case, 13 ids are split across 5 partitions as: `[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10], [11, 12]]`

The results of the lookup are concatenated into a dense tensor. The returned tensor has shape `shape(ids) + shape(params)[1:]`.

Args:

- `params`: A list of tensors with the same type and which can be concatenated along dimension 0. Each`Tensor` must be appropriately sized for the given `partition_strategy`.
- `ids`: A `Tensor` with type `int32` or `int64` containing the ids to be looked up in `params`.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(params) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`.
- `name`: A name for the operation (optional).
- `validate_indices`: Whether or not to validate gather indices.

Returns:

A `Tensor` with the same type as the tensors in `params`.

Raises:

- `ValueError`: If `params` is empty.

Evaluation

The evaluation ops are useful for measuring the performance of a network. Since they are nondifferentiable, they are typically used at evaluation time.

`tf.nn.top_k(input, k=1, sorted=True, name=None)`

Finds values and indices of the `k` largest entries for the last dimension.

If the input is a vector (rank-1), finds the `k` largest entries in the vector and outputs their values and indices as vectors. Thus `values[j]` is the `j`-th largest entry in `input`, and its index is `indices[j]`.

For matrices (resp. higher rank input), computes the top `k` entries in each row (resp. vector along the last dimension). Thus,

```
values.shape = indices.shape = input.shape[:-1] + [k]
```

If two elements are equal, the lower-index element appears first.

Args:

- `input`: 1-D or higher `Tensor` with last dimension at least `k`.
- `k`: 0-D `int32` `Tensor`. Number of top elements to look for along the last dimension (along each row for matrices).
- `sorted`: If true the resulting `k` elements will be sorted by the values in descending order.
- `name`: Optional name for the operation.

Returns:

- `values`: The `k` largest elements along each last dimensional slice.
- `indices`: The indices of `values` within the last dimension of `input`.

`tf.nn.in_top_k(predictions, targets, k, name=None)`

Says whether the targets are in the top `K` predictions.

This outputs a `batch_size` bool array, an entry `out[i]` is `true` if the prediction for the target class is among the top `k` predictions among all predictions for example `i`. Note that the behavior of `InTopK` differs from the `TopK`op in its handling of ties; if multiple classes have the same prediction value and straddle the top-`k` boundary, all of those classes are considered to be in the top `k`.

More formally, let

$predictions_i$ be the predictions for all classes for example `i`, $targets_i$ be the target class for example `i`, out_i be the output for example `i`,

$$out_i = predictions_{i, targets_i} \in TopKIncludingTies(predictions_i)$$

Args:

- `predictions`: A `Tensor` of type `float32`.

  A `batch_size` x `classes` tensor.
- `targets`: A `Tensor`. Must be one of the following types: `int32`, `int64`.

  A `batch_size` vector of class ids.
- `k`: An `int`. Number of top elements to look at for computing precision.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`. Computed Precision at `k` as a `bool Tensor`.

Candidate Sampling

Do you want to train a multiclass or multilabel model with thousands or millions of output classes (for example, a language model with a large vocabulary)? Training with a full Softmax is slow in this case, since all of the classes are evaluated for every training example. Candidate Sampling training algorithms can speed up your step times by only considering a small randomly-chosen subset of contrastive classes (called candidates) for each batch of training examples.

See our [Candidate Sampling Algorithms Reference]

Sampled Loss Functions

TensorFlow provides the following sampled loss functions for faster training.

```
tf.nn.nce_loss(weights, biases, inputs, labels,
num_sampled, num_classes, num_true=1,
sampled_values=None, remove_accidental_hits=False,
partition_strategy='mod', name='nce_loss')
```

Computes and returns the noise-contrastive estimation training loss.

See [Noise-contrastive estimation: A new estimation principle for unnormalized statistical models]. Also see our[Candidate Sampling Algorithms Reference]

Note: In the case where `num_true` > 1, we assign to each target class the target probability 1 / `num_true` so that the target probabilities sum to 1 per-example.

Note: It would be useful to allow a variable number of target classes per example. We hope to provide this functionality in a future release. For now, if you have a variable number of target classes, you can pad them out to a constant number by either repeating them or by padding with an otherwise unused class.

Args:

- `weights`: A `Tensor` of shape `[num_classes, dim]`, or a list of `Tensor` objects whose concatenation along dimension 0 has shape [num_classes, dim]. The (possibly-partitioned) class embeddings.
- `biases`: A `Tensor` of shape `[num_classes]`. The class biases.
- `inputs`: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- `labels`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `num_classes`: An `int`. The number of possible classes.
- `num_true`: An `int`. The number of target classes per training example.
- `sampled_values`: a tuple of (`sampled_candidates`, `true_expected_count`,`sampled_expected_count`) returned by a `*_candidate_sampler` function. (if None, we default to`log_uniform_candidate_sampler`)
- `remove_accidental_hits`: A `bool`. Whether to remove "accidental hits" where a sampled class equals one of the target classes. If set to `True`, this is a "Sampled Logistic" loss instead of NCE, and we are

learning to generate log-odds instead of log probabilities. See our Candidate Sampling Algorithms Reference. Default is False.

- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(weights) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name`: A name for the operation (optional).

Returns:

A `batch_size` 1-D tensor of per-example NCE losses.

```
tf.nn.sampled_softmax_loss(weights, biases, inputs,
labels, num_sampled, num_classes, num_true=1,
sampled_values=None, remove_accidental_hits=True,
partition_strategy='mod', name='sampled_softmax_loss')
```

Computes and returns the sampled softmax training loss.

This is a faster way to train a softmax classifier over a huge number of classes.

This operation is for training only. It is generally an underestimate of the full softmax loss.

At inference time, you can compute full softmax probabilities with the expression`tf.nn.softmax(tf.matmul(inputs, weights) + biases)`.

See our Candidate Sampling Algorithms Reference
Also see Section 3 of Jean et al., 2014 (pdf) for the math.

Args:

- `weights`: A `Tensor` of shape `[num_classes, dim]`, or a list of `Tensor` objects whose concatenation along dimension 0 has shape [num_classes, dim]. The (possibly-sharded) class embeddings.
- `biases`: A `Tensor` of shape `[num_classes]`. The class biases.
- `inputs`: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- `labels`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes. Note that this format differs from the `labels` argument of `nn.softmax_cross_entropy_with_logits`.
- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `num_classes`: An `int`. The number of possible classes.
- `num_true`: An `int`. The number of target classes per training example.
- `sampled_values`: a tuple of (`sampled_candidates`, `true_expected_count`,`sampled_expected_count`) returned by a `*_candidate_sampler` function. (if None, we default to`log_uniform_candidate_sampler`)
- `remove_accidental_hits`: A `bool`. whether to remove "accidental hits" where a sampled class equals one of the target classes. Default is True.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(weights) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name`: A name for the operation (optional).

Returns:

A `batch_size` 1-D tensor of per-example sampled softmax losses.

Candidate Samplers

TensorFlow provides the following samplers for randomly sampling candidate classes when using one of the sampled loss functions above.

`tf.nn.uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`

Samples a set of classes using a uniform base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without

replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.
The base distribution for this operation is the uniform distribution over the range of integers `[0, range_max]`.
In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in this document.

If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true`: An `int`. The number of target classes per training example.
- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `unique`: A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max`: An `int`. The number of possible classes.
- `seed`: An `int`. An operation-specific seed. Default is 0.
- `name`: A name for the operation (optional).

Returns:

- `sampled_candidates`: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count`: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count`: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.log_uniform_candidate_sampler(true_classes,
num_true, num_sampled, unique, range_max, seed=None,
name=None)
```

Samples a set of classes using a log-uniform (Zipfian) base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is an approximately log-uniform or Zipfian distribution:

```
P(class) = (log(class + 2) - log(class + 1)) /
log(range_max + 1)
```

This sampler is useful when the target classes approximately follow such a distribution - for example, if the classes represent words in a lexicon sorted in decreasing order of frequency. If your classes are not ordered by decreasing frequency, do not use this op.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in [this document].

If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true`: An `int`. The number of target classes per training example.
- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `unique`: A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max`: An `int`. The number of possible classes.
- `seed`: An `int`. An operation-specific seed. Default is 0.
- `name`: A name for the operation (optional).

Returns:

- `sampled_candidates`: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count`: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count`: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.learned_unigram_candidate_sampler(true_classes,
num_true, num_sampled, unique, range_max, seed=None,
name=None)
```

Samples a set of classes from a distribution learned during training.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.
The base distribution for this operation is constructed on the fly during training. It is a unigram distribution over the target classes seen so far during training. Every integer in `[0, range_max]` begins with a weight of 1, and is incremented by 1 each time it is seen as a target class. The base distribution is not saved to checkpoints, so it is reset when the model is reloaded.
In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in this document.

If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true`: An `int`. The number of target classes per training example.

- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `unique`: A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max`: An `int`. The number of possible classes.
- `seed`: An `int`. An operation-specific seed. Default is 0.
- `name`: A name for the operation (optional).

Returns:

- `sampled_candidates`: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count`: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count`: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.fixed_unigram_candidate_sampler(true_classes,
num_true, num_sampled, unique, range_max, vocab_file='',
distortion=1.0, num_reserved_ids=0, num_shards=1,
shard=0, unigrams=(), seed=None, name=None)
```

Samples a set of classes using the provided (fixed) base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max)`.

The elements of `sampled_candidates` are drawn without

replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution is read from a file or passed in as an in-memory array. There is also an option to skew the distribution by applying a distortion power to the weights.

In addition, this operation returns

tensors `true_expected_count` and `sampled_expected_count`represes enting the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in this document.

If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true`: An `int`. The number of target classes per training example.
- `num_sampled`: An `int`. The number of classes to randomly sample per batch.
- `unique`: A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max`: An `int`. The number of possible classes.

- `vocab_file`: Each valid line in this file (which should have a CSV-like format) corresponds to a valid word ID. IDs are in sequential order, starting from num_reserved_ids. The last entry in each line is expected to be a value corresponding to the count or relative probability. Exactly one of `vocab_file` and `unigrams` needs to be passed to this operation.
- `distortion`: The distortion is used to skew the unigram probability distribution. Each weight is first raised to the distortion's power before adding to the internal unigram distribution. As a result, `distortion = 1.0`gives regular unigram sampling (as defined by the vocab file), and `distortion = 0.0` gives a uniform distribution.
- `num_reserved_ids`: Optionally some reserved IDs can be added in the range `[0, num_reserved_ids]`by the users. One use case is that a special unknown word token is used as ID 0. These IDs will have a sampling probability of 0.
- `num_shards`: A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `shard`) indicates the number of partitions that are being used in the overall computation.
- `shard`: A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `num_shards`) indicates the particular partition number of the operation, when partitioning is being used.
- `unigrams`: A list of unigram counts or probabilities, one per ID in sequential order. Exactly one of`vocab_file` and `unigrams` should be passed to this operation.
- `seed`: An `int`. An operation-specific seed. Default is 0.
- `name`: A name for the operation (optional).

Returns:

- `sampled_candidates`: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count`: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count`: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

Miscellaneous candidate sampling utilities

`tf.nn.compute_accidental_hits(true_classes, sampled_candidates, num_true, seed=None, name=None)`

Compute the position ids in `sampled_candidates` matching `true_classes`.

In Candidate Sampling, this operation facilitates virtually removing sampled classes which happen to match target classes. This is done in Sampled Softmax and Sampled Logistic.

See our Candidate Sampling Algorithms Reference.

We presuppose that the `sampled_candidates` are unique.

We call it an 'accidental hit' when one of the target classes matches one of the sampled classes. This operation reports accidental hits as triples `(index, id, weight)`, where `index` represents the row number in`true_classes`, `id` represents the position

in `sampled_candidates`, and weight is `-FLOAT_MAX`.

The result of this op should be passed through

a `sparse_to_dense` operation, then added to the logits of the

sampled classes. This removes the contradictory effect of accidentally sampling the true target classes as noise classes for the same example.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `sampled_candidates`: A tensor of type `int64` and shape `[num_sampled]`. The sampled_candidates output of CandidateSampler.
- `num_true`: An `int`. The number of target classes per training example.
- `seed`: An `int`. An operation-specific seed. Default is 0.
- `name`: A name for the operation (optional).

Returns:

- `indices`: A `Tensor` of type `int32` and shape `[num_accidental_hits]`. Values indicate rows in`true_classes`.
- `ids`: A `Tensor` of type `int64` and shape `[num_accidental_hits]`. Values indicate positions in`sampled_candidates`.

* `weights`: A `Tensor` of type `float` and

  shape `[num_accidental_hits]`. Each value is `-FLOAT_MAX`.

Running Graphs

Contents

This library contains classes for launching graphs and executing operations.

The basic usage guide has examples of how a graph is launched in a `tf.Session`.

Session management

`class tf.Session`

A class for running TensorFlow operations.

A `Session` object encapsulates the environment in

which `Operation` objects are executed, and `Tensor` objects are evaluated. For example:

```
# Build a graph.
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `c`.
print(sess.run(c))
```

A session may own resources, such as variables, queues, and readers. It is important to release these resources when they are no longer required. To do this, either invoke the `close()` method on the session, or use the session as a context manager. The following two examples are equivalent:

```
# Using the `close()` method.
sess = tf.Session()
sess.run(...)
sess.close()

# Using the context manager.
with tf.Session() as sess:
  sess.run(...)
```

The `ConfigProto` protocol buffer exposes various configuration options for a session. For example, to create a session that uses soft constraints for device placement, and log the resulting placement decisions, create a session as follows:

```
# Launch the graph in a session that allows soft device placement
and
# logs the placement decisions.
```

```
sess = 
tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                 log_device_placement=True))
```

`tf.Session.__init__(target='', graph=None, config=None)`

Creates a new TensorFlow session.

If no `graph` argument is specified when constructing the session, the default graph will be launched in the session. If you are using more than one graph (created with `tf.Graph()` in the same process, you will have to use different sessions for each graph, but each graph can be used in multiple sessions. In this case, it is often clearer to pass the graph to be launched explicitly to the session constructor.

Args:

- `target`: (Optional.) The execution engine to connect to. Defaults to using an in-process engine. At present, no value other than the empty string is supported.
- `graph`: (Optional.) The `Graph` to be launched (described above).
- `config`: (Optional.) A `ConfigProto` protocol buffer with configuration options for the session.

`tf.Session.run(fetches, feed_dict=None)`

Runs the operations and evaluates the tensors in `fetches`.

This method runs one "step" of TensorFlow computation, by running the necessary graph fragment to execute every `Operation` and evaluate every `Tensor` in `fetches`, substituting the values in `feed_dict` for the corresponding input values.

The `fetches` argument may be a list of graph elements or a single graph element, and these determine the return value of this method. A graph element can be one of the following types:

- If the *i*th element of `fetches` is an `Operation`, the *i*th return value will be `None`.
- If the *i*th element of `fetches` is a `Tensor`, the *i*th return value will be a numpy ndarray containing the value of that tensor.
- If the *i*th element of `fetches` is a `SparseTensor`, the *i*th return value will be a `SparseTensorValue`containing the value of that sparse tensor.

The optional `feed_dict` argument allows the caller to override the value of tensors in the graph. Each key in`feed_dict` can be one of the following types:

- If the key is a `Tensor`, the value may be a Python scalar, string, list, or numpy ndarray that can be converted to the same `dtype` as that tensor. Additionally, if the key is a placeholder, the shape of the value will be checked for compatibility with the placeholder.
- If the key is a `SparseTensor`, the value should be a `SparseTensorValue`.

Args:

- `fetches`: A single graph element, or a list of graph elements (described above).

* `feed_dict`: A dictionary that maps graph elements to values (described above).

Returns:

Either a single value if `fetches` is a single graph element, or a list of values if `fetches` is a list (described above).

Raises:

* `RuntimeError`: If this `Session` is in an invalid state (e.g. has been closed).
* `TypeError`: If `fetches` or `feed_dict` keys are of an inappropriate type.
* `ValueError`: If `fetches` or `feed_dict` keys are invalid or refer to a `Tensor` that doesn't exist.

`tf.Session.close()`

Closes this session.

Calling this method frees all resources associated with the session.

Raises:

* `RuntimeError`: If an error occurs while closing the session.

`tf.Session.graph`

The graph that was launched in this session.

`tf.Session.as_default()`

Returns a context manager that makes this object the default session.

Use with the `with` keyword to specify that calls

to `Operation.run()` or `Tensor.run()` should be executed in this

session.

```
c = tf.constant(..)
sess = tf.Session()

with sess.as_default():
  assert tf.get_default_session() is sess
  print(c.eval())
```

To get the current default session, use `tf.get_default_session()`.

N.B. The `as_default` context manager does not close the session

when you exit the context, and you must close the session explicitly.

```
c = tf.constant(...)
sess = tf.Session()
with sess.as_default():
  print(c.eval())
# ...
with sess.as_default():
  print(c.eval())

sess.close()
```

Alternatively, you can use `with tf.Session():` to create a session

that is automatically closed on exiting the context, including when an uncaught exception is raised.
N.B. The default graph is a property of the current thread. If you create a new thread, and wish to use the default session in that

thread, you must explicitly add a `with sess.as_default():` in that thread's function.

Returns:

A context manager using this session as the default session.

`class tf.InteractiveSession`

A TensorFlow `Session` for use in interactive contexts, such as a shell.

The only difference with a regular `Session` is that

an `InteractiveSession` installs itself as the default session on construction. The methods `Tensor.eval()` and `Operation.run()` will use that session to run ops.

This is convenient in interactive shells and [IPython notebooks](), as it avoids having to pass an explicit `Session`object to run ops.

For example:

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# We can just use 'c.eval()' without passing 'sess'
print(c.eval())
sess.close()
```

Note that a regular session installs itself as the default session when it is created in a `with` statement. The common usage in non-interactive programs is to follow that pattern:

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
```

```
with tf.Session():
  # We can also use 'c.eval()' here.
  print(c.eval())
```

`tf.InteractiveSession.__init__(target='', graph=None, config=None)`

Creates a new interactive TensorFlow session.

If no `graph` argument is specified when constructing the session, the default graph will be launched in the session. If you are using more than one graph (created with `tf.Graph()` in the same process, you will have to use different sessions for each graph, but each graph can be used in multiple sessions. In this case, it is often clearer to pass the graph to be launched explicitly to the session constructor.

Args:

- `target`: (Optional.) The execution engine to connect to. Defaults to using an in-process engine. At present, no value other than the empty string is supported.
- `graph`: (Optional.) The `Graph` to be launched (described above).
- `config`: (Optional) `ConfigProto` proto used to configure the session.

`tf.InteractiveSession.close()`

Closes an `InteractiveSession`.

`tf.get_default_session()`

Returns the default session for the current thread.

The returned `Session` will be the innermost session on which a `Session` or `Session.as_default()` context has been entered.

NOTE: The default session is a property of the current thread. If you create a new thread, and wish to use the default session in that thread, you must explicitly add a `with sess.as_default():` in that thread's function.

Returns:

The default `Session` being used in the current thread.

Error classes

`class tf.OpError`

A generic error that is raised when TensorFlow execution fails.

Whenever possible, the session will raise a more specific subclass of `OpError` from the `tf.errors` module.

`tf.OpError.op`

The operation that failed, if known.

N.B. If the failed op was synthesized at runtime, e.g. a `Send` or `Recv` op, there will be no corresponding`Operation` object. In that case, this will return `None`, and you should instead use the `OpError.node_def` to discover information about the op.

Returns:

The `Operation` that failed, or None.

`tf.OpError.node_def`

The `NodeDef` proto representing the op that failed.

Other Methods

`tf.OpError.__init__(node_def, op, message, error_code)`

Creates a new `OpError` indicating that a particular op failed.

Args:

- `node_def`: The `graph_pb2.NodeDef` proto representing the op that failed.
- `op`: The `ops.Operation` that failed, if known; otherwise None.
- `message`: The message string describing the failure.
- `error_code`: The `error_codes_pb2.Code` describing the error.

`tf.OpError.error_code`

The integer error code that describes the error.

`tf.OpError.message`

The error message that describes the error.

`class tf.errors.CancelledError`

Raised when an operation or step is cancelled.

For example, a long-running operation (e.g. `queue.enqueue()` may be cancelled by running another operation (e.g. `queue.close(cancel_pending_enqueues=True)`, or by closing the session. A step that is running such a long-running operation will fail by raising `CancelledError`.

`tf.errors.CancelledError.__init__(node_def, op, message)`

Creates a `CancelledError`.

`class tf.errors.UnknownError`

Unknown error.

An example of where this error may be returned is if a Status value received from another address space belongs to an error-space that is not known to this address space. Also errors raised by APIs that do not return enough error information may be converted to this error.

`tf.errors.UnknownError.__init__(node_def, op, message, error_code=2)`

Creates an `UnknownError`.

`class tf.errors.InvalidArgumentError`

Raised when an operation receives an invalid argument.

This may occur, for example, if an operation is receives an input tensor that has an invalid value or shape. For example, the `tf.matmul()` op will raise this error if it receives an input that is not a matrix, and the `tf.reshape()` op will raise this error if the new shape does not match the number of elements in the input tensor.

`tf.errors.InvalidArgumentError.__init__(node_def, op, message)`

Creates an `InvalidArgumentError`.

`class tf.errors.DeadlineExceededError`

Raised when a deadline expires before an operation could complete.

This exception is not currently used.

`tf.errors.DeadlineExceededError.__init__(node_def, op, message)`

Creates a `DeadlineExceededError`.

`class tf.errors.NotFoundError`

Raised when a requested entity (e.g., a file or directory) was not found.

For example, running the `tf.WholeFileReader.read()` operation could raise `NotFoundError` if it receives the name of a file that does not exist.

`tf.errors.NotFoundError.__init__(node_def, op, message)`

Creates a `NotFoundError`.

`class tf.errors.AlreadyExistsError`

Raised when an entity that we attempted to create already exists.

For example, running an operation that saves a file (e.g. `tf.train.Saver.save()`) could potentially raise this exception if an explicit filename for an existing file was passed.

`tf.errors.AlreadyExistsError.__init__(node_def, op, message)`

Creates an `AlreadyExistsError`.

`class tf.errors.PermissionDeniedError`

Raised when the caller does not have permission to run an operation.

For example, running the `tf.WholeFileReader.read()` operation could raise `PermissionDeniedError`if it receives the name of a file for which the user does not have the read file permission.

`tf.errors.PermissionDeniedError.__init__(node_def, op, message)`

Creates a `PermissionDeniedError`.

`class tf.errors.UnauthenticatedError`

The request does not have valid authentication credentials.

This exception is not currently used.

`tf.errors.UnauthenticatedError.__init__(node_def, op, message)`

Creates an `UnauthenticatedError`.

`class tf.errors.ResourceExhaustedError`

Some resource has been exhausted.

For example, this error might be raised if a per-user quota is exhausted, or perhaps the entire file system is out of space.

`tf.errors.ResourceExhaustedError.__init__(node_def, op, message)`

Creates a `ResourceExhaustedError`.

`class tf.errors.FailedPreconditionError`

Operation was rejected because the system is not in a state to execute it.

This exception is most commonly raised when running an operation that reads a `tf.Variable` before it has been initialized.

`tf.errors.FailedPreconditionError.__init__(node_def, op, message)`

Creates a `FailedPreconditionError`.

`class tf.errors.AbortedError`

The operation was aborted, typically due to a concurrent action.

For example, running a `queue.enqueue()` operation may

raise `AbortedError` if a `queue.close()` operation previously ran.

`tf.errors.AbortedError.__init__(node_def, op, message)`

Creates an `AbortedError`.

`class tf.errors.OutOfRangeError`

Raised when an operation executed past the valid range.

This exception is raised in "end-of-file" conditions, such as when a `queue.dequeue()` operation is blocked on an empty queue, and

a `queue.close()` operation executes.

`tf.errors.OutOfRangeError.__init__(node_def, op, message)`

Creates an `OutOfRangeError`.

`class tf.errors.UnimplementedError`

Raised when an operation has not been implemented.

Some operations may raise this error when passed otherwise-valid arguments that it does not currently support. For example, running the `tf.nn.max_pool()` operation would raise this error if pooling was requested on the batch dimension, because this is not yet supported.

`tf.errors.UnimplementedError.__init__(node_def, op, message)`

Creates an `UnimplementedError`.

`class tf.errors.InternalError`

Raised when the system experiences an internal error.

This exception is raised when some invariant expected by the runtime has been broken. Catching this exception is not recommended.

`tf.errors.InternalError.__init__(node_def, op, message)`

Creates an `InternalError`.

`class tf.errors.UnavailableError`

Raised when the runtime is currently unavailable.

This exception is not currently used.

`tf.errors.UnavailableError.__init__(node_def, op, message)`

Creates an `UnavailableError`.

`class tf.errors.DataLossError`

Raised when unrecoverable data loss or corruption is encountered.

For example, this may be raised by running a `tf.WholeFileReader.read()` operation, if the file is truncated while it is being read.

`tf.errors.DataLossError.__init__(node_def, op, message)`

Creates a `DataLossError`.

Training

Contents

- `tf.image_summary(tag, tensor, max_images=3, collections=None, name=None)`
- `tf.histogram_summary(tag, values, collections=None, name=None)`
- `tf.nn.zero_fraction(value, name=None)`
- `tf.merge_summary(inputs, collections=None, name=None)`
- `tf.merge_all_summaries(key=summaries)`
- Adding Summaries to Event Files
- `class tf.train.SummaryWriter`
- `tf.train.summary_iterator(path)`
- Training utilities
- `tf.train.global_step(sess, global_step_tensor)`
- `tf.train.write_graph(graph_def, logdir, name, as_text=True)`
- Other Functions and Classes
- `class tf.train.LooperThread`
- `tf.train.export_meta_graph(filename=None, meta_info_def=None, graph_def=None, saver_def=None, collection_list=None, as_text=False)`
- `tf.train.generate_checkpoint_state_proto(save_dir, model_checkpoint_path, all_model_checkpoint_paths=None)`
- `tf.train.import_meta_graph(meta_graph_or_file)`

This library provides a set of classes and functions that helps train models.

Optimizers

The Optimizer base class provides methods to compute gradients for a loss and apply gradients to variables. A collection of subclasses implement classic optimization algorithms such as GradientDescent and Adagrad.

You never instantiate the Optimizer class itself, but instead instantiate one of the subclasses.

`class tf.train.Optimizer`

Base class for optimizers.

This class defines the API to add Ops to train a model. You never use this class directly, but instead instantiate one of its subclasses such as `GradientDescentOptimizer`, `AdagradOptimizer`, or `MomentumOptimizer`.

Usage

```
# Create an optimizer with the desired parameters.
opt = GradientDescentOptimizer(learning_rate=0.1)
# Add Ops to the graph to minimize a cost by updating a list of
variables.
# "cost" is a Tensor, and the list of variables contains
tf.Variable
# objects.
opt_op = opt.minimize(cost, var_list=<list of variables>)
```

In the training program you will just have to run the returned Op.

```
# Execute opt_op to do one step of training:
opt_op.run()
```

Processing gradients before applying them.

Calling `minimize()` takes care of both computing the gradients and applying them to the variables. If you want to process the gradients before applying them you can instead use the optimizer in three steps:

1. Compute the gradients with `compute_gradients()`.
2. Process the gradients as you wish.
3. Apply the processed gradients with `apply_gradients()`.

Example:

```
# Create an optimizer.
opt = GradientDescentOptimizer(learning_rate=0.1)

# Compute the gradients for a list of variables.
```

```
grads_and_vars = opt.compute_gradients(loss, <list of variables>)

# grads_and_vars is a list of tuples (gradient, variable).  Do
whatever you
# need to the 'gradient' part, for example cap them, etc.
capped_grads_and_vars = [(MyCapper(gv[0]), gv[1])) for gv in
grads_and_vars]

# Ask the optimizer to apply the capped gradients.
opt.apply_gradients(capped_grads_and_vars)
```

`tf.train.Optimizer.__init__(use_locking, name)`

Create a new Optimizer.

This must be called by the constructors of subclasses.

Args:

- `use_locking`: Bool. If True apply use locks to prevent concurrent updates to variables.
- `name`: A non-empty string. The name to use for accumulators created for the optimizer.

Raises:

- `ValueError`: If name is malformed.

`tf.train.Optimizer.minimize(loss, global_step=None, var_list=None, gate_gradients=1, aggregation_method=None, colocate_gradients_with_ops=False, name=None)`

Add operations to minimize `loss` by updating `var_list`.
This method simply combines calls `compute_gradients()` and `apply_gradients()`. If you want to process the gradient before applying them call `compute_gradients()` and `apply_gradients()` explicitly instead of using this function.

Args:

- `loss`: A `Tensor` containing the value to minimize.
- `global_step`: Optional `Variable` to increment by one after the variables have been updated.
- `var_list`: Optional list of `Variable` objects to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.
- `gate_gradients`: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or`GATE_GRAPH`.
- `aggregation_method`: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.
- `name`: Optional name for the returned operation.

Returns:

An Operation that updates the variables in `var_list`.

If `global_step` was not `None`, that operation also increments `global_step`.

Raises:

- `ValueError`: If some of the variables are not `Variable` objects.

```
tf.train.Optimizer.compute_gradients(loss, var_list=None,
gate_gradients=1, aggregation_method=None,
colocate_gradients_with_ops=False)
```

Compute gradients of `loss` for the variables in `var_list`.

This is the first part of `minimize()`. It returns a list of (gradient, variable) pairs where "gradient" is the gradient for "variable". Note that "gradient" can be a `Tensor`, an `IndexedSlices`, or `None` if there is no gradient for the given variable.

Args:

- `loss`: A Tensor containing the value to minimize.
- `var_list`: Optional list of tf.Variable to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKey.TRAINABLE_VARIABLES`.
- `gate_gradients`: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or`GATE_GRAPH`.

- `aggregation_method`: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.

Returns:

A list of (gradient, variable) pairs.

Raises:

- `TypeError`: If `var_list` contains anything else than `Variable` objects.
- `ValueError`: If some arguments are invalid.

`tf.train.Optimizer.apply_gradients(grads_and_vars, global_step=None, name=None)`

Apply gradients to variables.

This is the second part of `minimize()`. It returns an `Operation` that applies gradients.

Args:

- `grads_and_vars`: List of (gradient, variable) pairs as returned by `compute_gradients()`.

- `global_step`: Optional `Variable` to increment by one after the variables have been updated.
- `name`: Optional name for the returned operation. Default to the name passed to the `Optimizer` constructor.

Returns:

An `Operation` that applies the specified gradients.

If `global_step` was not None, that operation also

increments `global_step`.

Raises:

- `TypeError`: If `grads_and_vars` is malformed.
- `ValueError`: If none of the variables have gradients.

Gating Gradients

Both `minimize()` and `compute_gradients()` accept

a `gate_gradient` argument that controls the degree of parallelism during the application of the gradients.

The possible values are: `GATE_NONE`, `GATE_OP`, and `GATE_GRAPH`.

`GATE_NONE`: Compute and apply gradients in parallel. This provides the maximum parallelism in execution, at the cost of some non-reproducibility in the results. For example the two gradients

of `matmul` depend on the input values: With `GATE_NONE` one of the gradients could be applied to one of the inputs <u>before</u> the other gradient is computed resulting in non-reproducible results.

`GATE_OP`: For each Op, make sure all gradients are computed before they are used. This prevents race conditions for Ops that generate gradients for multiple inputs where the gradients depend on the inputs.

`GATE_GRAPH`: Make sure all gradients for all variables are computed before any one of them is used. This provides the least parallelism but can be useful if you want to process all gradients before applying any of them.

Slots

Some optimizer subclasses, such as `MomentumOptimizer` and `AdagradOptimizer` allocate and manage additional variables associated with the variables to train. These are called *Slots*. Slots have names and you can ask the optimizer for the names of the slots that it uses. Once you have a slot name you can ask the optimizer for the variable it created to hold the slot value.

This can be useful if you want to log debug a training algorithm, report stats about the slots, etc.

`tf.train.Optimizer.get_slot_names()`

Return a list of the names of slots created by the `Optimizer`.

See `get_slot()`.

Returns:

A list of strings.

`tf.train.Optimizer.get_slot(var, name)`

Return a slot named `name` created for `var` by the Optimizer.

Some `Optimizer` subclasses use additional variables. For example `Momentum` and `Adagrad` use variables to accumulate updates. This method gives access to these `Variable` objects if for some reason you need them.

Use `get_slot_names()` to get the list of slot names created by the `Optimizer`.

Args:

- `var`: A variable passed to `minimize()` or `apply_gradients()`.
- `name`: A string.

Returns:

The `Variable` for the slot if it was created, `None` otherwise.

`class tf.train.GradientDescentOptimizer`

Optimizer that implements the gradient descent algorithm.

`tf.train.GradientDescentOptimizer.__init__(learning_rate, use_locking=False, name='GradientDescent')`

Construct a new gradient descent optimizer.

Args:

* `learning_rate`: A Tensor or a floating point value. The learning rate to use.
* `use_locking`: If True use locks for update operations.
* `name`: Optional name prefix for the operations created when applying gradients. Defaults to "GradientDescent".

`class tf.train.AdagradOptimizer`

Optimizer that implements the Adagrad algorithm.

See this paper.

```
tf.train.AdagradOptimizer.__init__(learning_rate,
initial_accumulator_value=0.1, use_locking=False,
name='Adagrad')
```

Construct a new Adagrad optimizer.

Args:

* `learning_rate`: A `Tensor` or a floating point value. The learning rate.
* `initial_accumulator_value`: A floating point value. Starting value for the accumulators, must be positive.
* `use_locking`: If `True` use locks for update operations.
* `name`: Optional name prefix for the operations created when applying gradients. Defaults to "Adagrad".

Raises:

- `ValueError`: If the `initial_accumulator_value` is invalid.

`class tf.train.MomentumOptimizer`

Optimizer that implements the Momentum algorithm.

`tf.train.MomentumOptimizer.__init__(learning_rate, momentum, use_locking=False, name='Momentum')`

Construct a new Momentum optimizer.

Args:

- `learning_rate`: A `Tensor` or a floating point value. The learning rate.
- `momentum`: A `Tensor` or a floating point value. The momentum.
- `use_locking`: If `True` use locks for update operations.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "Momentum".

`class tf.train.AdamOptimizer`

Optimizer that implements the Adam algorithm.

See Kingma et. al., 2014 (pdf).

`tf.train.AdamOptimizer.__init__(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False, name='Adam')`

Construct a new Adam optimizer.

Initialization:

```
m_0 <- 0 (Initialize initial 1st moment vector)
v_0 <- 0 (Initialize initial 2nd moment vector)
t <- 0 (Initialize timestep)
```

The update rule for `variable` with gradient `g` uses an optimization described at the end of section2 of the paper:

```
t <- t + 1
lr_t <- learning_rate * sqrt(1 - beta2^t) / (1 - beta1^t)

m_t <- beta1 * m_{t-1} + (1 - beta1) * g
v_t <- beta2 * v_{t-1} + (1 - beta2) * g * g
variable <- variable - lr_t * m_t / (sqrt(v_t) + epsilon)
```

The default value of 1e-8 for epsilon might not be a good default in general. For example, when training an Inception network on ImageNet a current good choice is 1.0 or 0.1.

Args:

- `learning_rate`: A Tensor or a floating point value. The learning rate.
- `beta1`: A float value or a constant float tensor. The exponential decay rate for the 1st moment estimates.
- `beta2`: A float value or a constant float tensor. The exponential decay rate for the 2nd moment estimates.
- `epsilon`: A small constant for numerical stability.
- `use_locking`: If True use locks for update operations.

- `name`: Optional name for the operations created when applying gradients. Defaults to "Adam".

`class tf.train.FtrlOptimizer`

Optimizer that implements the FTRL algorithm.

`tf.train.FtrlOptimizer.__init__(learning_rate, learning_rate_power=-0.5, initial_accumulator_value=0.1, l1_regularization_strength=0.0, l2_regularization_strength=0.0, use_locking=False, name='Ftrl')`

Construct a new FTRL optimizer.

The Ftrl-proximal algorithm, abbreviated for Follow-the-regularized-leader, is described in the paper [Ad Click Prediction: a View from the Trenches]().

It can give a good performance vs. sparsity tradeoff.

Ftrl-proximal uses its own global base learning rate and can behave like Adagrad with`learning_rate_power=-0.5`, or like gradient descent with `learning_rate_power=0.0`.

The effective learning rate is adjusted per parameter, relative to this base learning rate as:

```
effective_learning_rate_i = (learning_rate /
    pow(k + summed_squared_gradients_for_i, learning_rate_power));
```

where k is the small constant `initial_accumulator_value`.

Note that the real regularization coefficient of `|w|^2` for objective function is `1 / lambda_2` if specifying `l2 = lambda_2` as argument when using this function.

Args:

- `learning_rate`: A float value or a constant float `Tensor`.
- `learning_rate_power`: A float value, must be less or equal to zero.
- `initial_accumulator_value`: The starting value for accumulators. Only positive values are allowed.
- `l1_regularization_strength`: A float value, must be greater than or equal to zero.
- `l2_regularization_strength`: A float value, must be greater than or equal to zero.
- `use_locking`: If `True` use locks for update operations.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "Ftrl".

Raises:

- `ValueError`: If one of the arguments is invalid.

`class tf.train.RMSPropOptimizer`

Optimizer that implements the RMSProp algorithm.

See the paper.

```
tf.train.RMSPropOptimizer.__init__(learning_rate, 
decay=0.9, momentum=0.0, epsilon=1e-10, 
use_locking=False, name='RMSProp')
```

Construct a new RMSProp optimizer.

Args:

- `learning_rate`: A Tensor or a floating point value. The learning rate.
- `decay`: Discounting factor for the history/coming gradient
- `momentum`: A scalar tensor.
- `epsilon`: Small value to avoid zero denominator.
- `use_locking`: If True use locks for update operation.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "RMSProp".

Gradient Computation

TensorFlow provides functions to compute the derivatives for a given TensorFlow computation graph, adding operations to the graph. The optimizer classes automatically compute derivatives on your graph, but creators of new Optimizers or expert users can call the lower-level functions below.

```
tf.gradients(ys, xs, grad_ys=None, name='gradients', 
colocate_gradients_with_ops=False, gate_gradients=False, 
aggregation_method=None)
```

Constructs symbolic partial derivatives of `ys` w.r.t. x in `xs`.

`ys` and `xs` are each a `Tensor` or a list of tensors. `grad_ys` is a list of `Tensor`, holding the gradients received by the `ys`. The list must be the same length as `ys`.

`gradients()` adds ops to the graph to output the partial derivatives of `ys` with respect to `xs`. It returns a list of`Tensor` of length `len(xs)` where each tensor is the `sum(dy/dx)` for y in `ys`.

`grad_ys` is a list of tensors of the same length as `ys` that holds the initial gradients for each y in `ys`. When`grad_ys` is None, we fill in a tensor of '1's of the shape of y for each y in `ys`. A user can provide their own initial`grad_ys` to compute the derivatives using a different initial gradient for each y (e.g., if one wanted to weight the gradient differently for each value in each y).

Args:

- `ys`: A `Tensor` or list of tensors to be differentiated.
- `xs`: A `Tensor` or list of tensors to be used for differentiation.
- `grad_ys`: Optional. A `Tensor` or list of tensors the same size as `ys` and holding the gradients computed for each y in `ys`.
- `name`: Optional name to use for grouping all the gradient ops together. defaults to 'gradients'.
- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.
- `gate_gradients`: If True, add a tuple around the gradients returned for an operations. This avoids some race conditions.

* `aggregation_method`: Specifies the method used to combine gradient terms. Accepted values are constants defined in the class `AggregationMethod`.

Returns:

A list of `sum(dy/dx)` for each x in `xs`.

Raises:

* `LookupError`: if one of the operations between `x` and `y` does not have a registered gradient function.
* `ValueError`: if the arguments are invalid.

`class tf.AggregationMethod`

A class listing aggregation methods used to combine gradients.

Computing partial derivatives can require aggregating gradient contributions. This class lists the various methods that can be used to combine gradients in the graph:

* `ADD_N`: All of the gradient terms are summed as part of one operation using the "AddN" op. It has the property that all gradients must be ready before any aggregation is performed.
* `DEFAULT`: The system-chosen default aggregation method.

`tf.stop_gradient(input, name=None)`

Stops gradient computation.

When executed in a graph, this op outputs its input tensor as-is.

When building ops to compute gradients, this op prevents the contribution of its inputs to be taken into account. Normally, the gradient generator adds ops to a graph to compute the derivatives of a specified 'loss' by recursively finding out inputs that contributed to its computation. If you insert this op in the graph it inputs are masked from the gradient generator. They are not taken into account for computing gradients.

This is useful any time you want to compute a value with TensorFlow but need to pretend that the value was a constant. Some examples include:

- The EM algorithm where the M-step should not involve backpropagation through the output of the E-step.
- Contrastive divergence training of Boltzmann machines where, when differentiating the energy function, the training must not backpropagate through the graph that generated the samples from the model.
- Adversarial training, where no backprop should happen through the adversarial example generation process.

Args:

- `input`: A `Tensor`.
- `name`: A name for the operation (optional).

Returns:

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

Gradient Clipping

TensorFlow provides several operations that you can use to add clipping functions to your graph. You can use these functions to perform general data clipping, but they're particularly useful for handling exploding or vanishing gradients.

`tf.clip_by_value(t, clip_value_min, clip_value_max, name=None)`

Clips tensor values to a specified min and max.

Given a tensor `t`, this operation returns a tensor of the same type and shape as `t` with its values clipped to`clip_value_min` and `clip_value_max`. Any values less than `clip_value_min` are set to`clip_value_min`. Any values greater than `clip_value_max` are set to `clip_value_max`.

Args:

- `t`: A `Tensor`.
- `clip_value_min`: A 0-D (scalar) `Tensor`. The minimum value to clip by.
- `clip_value_max`: A 0-D (scalar) `Tensor`. The maximum value to clip by.
- `name`: A name for the operation (optional).

Returns:

A clipped `Tensor`.

```
tf.clip_by_norm(t, clip_norm, name=None)
```

Clips tensor values to a maximum L2-norm.

Given a tensor `t`, and a maximum clip value `clip_norm`, this operation normalizes `t` so that its L2-norm is less than or equal to `clip_norm`. Specifically, if the L2-norm is already less than or equal to `clip_norm`, then `t` is not modified. If the L2-norm is greater than `clip_norm`, then this operation returns a tensor of the same type and shape as `t` with its values set to:

```
t * clip_norm / l2norm(t)
```

In this case, the L2-norm of the output tensor is `clip_norm`.

This operation is typically used to clip gradients before applying them with an optimizer.

Args:

- `t`: A `Tensor`.
- `clip_norm`: A 0-D (scalar) `Tensor` > 0. A maximum clipping value.
- `name`: A name for the operation (optional).

Returns:

A clipped `Tensor`.

`tf.clip_by_average_norm(t, clip_norm, name=None)`

Clips tensor values to a maximum average L2-norm.

Given a tensor `t`, and a maximum clip value `clip_norm`, this operation normalizes `t` so that its average L2-norm is less than or equal to `clip_norm`. Specifically, if the average L2-norm is already less than or equal to`clip_norm`, then `t` is not modified. If the average L2-norm is greater than `clip_norm`, then this operation returns a tensor of the same type and shape as `t` with its values set to:

`t * clip_norm / l2norm_avg(t)`

In this case, the average L2-norm of the output tensor is `clip_norm`.

This operation is typically used to clip gradients before applying them with an optimizer.

Args:

- `t`: A `Tensor`.
- `clip_norm`: A 0-D (scalar) `Tensor` > 0. A maximum clipping value.
- `name`: A name for the operation (optional).

Returns:

A clipped `Tensor`.

`tf.clip_by_global_norm(t_list, clip_norm, use_norm=None, name=None)`

Clips values of multiple tensors by the ratio of the sum of their norms.

Given a tuple or list of tensors `t_list`, and a clipping ratio `clip_norm`, this operation returns a list of clipped tensors `list_clipped` and the global norm (`global_norm`) of all tensors in `t_list`. Optionally, if you've already computed the global norm for `t_list`, you can specify the global norm with `use_norm`.

To perform the clipping, the values `t_list[i]` are set to:

```
t_list[i] * clip_norm / max(global_norm, clip_norm)
```

where:

```
global_norm = sqrt(sum([l2norm(t)**2 for t in t_list]))
```

If `clip_norm > global_norm` then the entries in `t_list` remain as they are, otherwise they're all shrunk by the global ratio.

Any of the entries of `t_list` that are of type `None` are ignored.

This is the correct way to perform gradient clipping (for example, see Pascanu et al., 2012 (pdf)).

However, it is slower than `clip_by_norm()` because all the parameters must be ready before the clipping operation can be performed.

Args:

- `t_list`: A tuple or list of mixed `Tensors`, `IndexedSlices`, or None.
- `clip_norm`: A 0-D (scalar) `Tensor` > 0. The clipping ratio.
- `use_norm`: A 0-D (scalar) `Tensor` of type `float` (optional). The global norm to use. If not provided,`global_norm()` is used to compute the norm.
- `name`: A name for the operation (optional).

Returns:

- `list_clipped`: A list of `Tensors` of the same type as `list_t`.
- `global_norm`: A 0-D (scalar) `Tensor` representing the global norm.

Raises:

- `TypeError`: If `t_list` is not a sequence.

`tf.global_norm(t_list, name=None)`

Computes the global norm of multiple tensors.

Given a tuple or list of tensors `t_list`, this operation returns the global norm of the elements in all tensors in`t_list`. The global norm is computed as:

```
global_norm = sqrt(sum([l2norm(t)**2 for t in t_list]))
```

Any entries in `t_list` that are of type None are ignored.

Args:

- `t_list`: A tuple or list of mixed `Tensors`, `IndexedSlices`, or None.
- `name`: A name for the operation (optional).

Returns:

A 0-D (scalar) `Tensor` of type `float`.

Raises:

- `TypeError`: If `t_list` is not a sequence.

Decaying the learning rate

`tf.train.exponential_decay(learning_rate, global_step, decay_steps, decay_rate, staircase=False, name=None)`

Applies exponential decay to the learning rate.

When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies an exponential decay function to a provided initial learning rate. It requires a `global_step` value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The function returns the decayed learning rate. It is computed as:

```
decayed_learning_rate = learning_rate *
                        decay_rate ^ (global_step / decay_steps)
```

If the argument `staircase` is `True`, then `global_step /decay_steps` is an integer division and the decayed learning rate follows a staircase function.

Example: decay every 100000 steps with a base of 0.96:

```
...
global_step = tf.Variable(0, trainable=False)
starter_learning_rate = 0.1
learning_rate = tf.train.exponential_decay(starter_learning_rate, global_step,
                                           100000, 0.96, staircase=True)
optimizer = tf.GradientDescentOptimizer(learning_rate)
```

```
# Passing global_step to minimize() will increment it at each step.
optimizer.minimize(...my loss..., global_step=global_step)
```

Args:

- `learning_rate`: A scalar `float32` or `float64` `Tensor` or a Python number. The initial learning rate.
- `global_step`: A scalar `int32` or `int64` `Tensor` or a Python number. Global step to use for the decay computation. Must not be negative.
- `decay_steps`: A scalar `int32` or `int64` `Tensor` or a Python number. Must be positive. See the decay computation above.
- `decay_rate`: A scalar `float32` or `float64` `Tensor` or a Python number. The decay rate.
- `staircase`: Boolean. It `True` decay the learning rate at discrete intervals.
- `name`: String. Optional name of the operation. Defaults to 'ExponentialDecay'

Returns:

A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Moving Averages

Some training algorithms, such as GradientDescent and Momentum often benefit from maintaining a moving average of variables during optimization. Using the moving averages for evaluations often improve results significantly.

`class tf.train.ExponentialMovingAverage`

Maintains moving averages of variables by employing an exponential decay.

When training a model, it is often beneficial to maintain moving averages of the trained parameters. Evaluations that use averaged parameters sometimes produce significantly better results than the final trained values.

The `apply()` method adds shadow copies of trained variables and add ops that maintain a moving average of the trained variables in their shadow copies. It is used when building the training model. The ops that maintain moving averages are typically run after each training step. The `average()` and `average_name()` methods give access to the shadow variables and their names. They are useful when building an evaluation model, or when restoring a model from a checkpoint file. They help use the moving averages in place of the last trained values for evaluations.
The moving averages are computed using exponential decay. You specify the decay value when creating the`ExponentialMovingAverage` object. The shadow variables are initialized with the same initial values as the trained variables. When you run the ops to maintain the moving averages, each shadow variable is updated with the formula:
`shadow_variable -= (1 - decay) * (shadow_variable - variable)`
This is mathematically equivalent to the classic formula below, but the use of an `assign_sub` op (the `"-="` in the formula) allows concurrent lockless updates to the variables:
`shadow_variable = decay * shadow_variable + (1 - decay) * variable`

Reasonable values for `decay` are close to 1.0, typically in the multiple-nines range: 0.999, 0.9999, etc.

Example usage when creating a training model:

```
# Create variables.
var0 = tf.Variable(...)
var1 = tf.Variable(...)
```

```
# ... use the variables to build a training model...
...
# Create an op that applies the optimizer.  This is what we
usually
# would use as a training op.
opt_op = opt.minimize(my_loss, [var0, var1])

# Create an ExponentialMovingAverage object
ema = tf.train.ExponentialMovingAverage(decay=0.9999)

# Create the shadow variables, and add ops to maintain moving
averages
# of var0 and var1.
maintain_averages_op = ema.apply([var0, var1])

# Create an op that will update the moving averages after each
training
# step.  This is what we will use in place of the usual training
op.
with tf.control_dependencies([opt_op]):
    training_op = tf.group(maintain_averages_op)

...train the model by running training_op...
```

There are two ways to use the moving averages for evaluations:

- Build a model that uses the shadow variables instead of the variables. For this, use the `average()` method which returns the shadow variable for a given variable.
- Build a model normally but load the checkpoint files to evaluate by using the shadow variable names. For this use the `average_name()` method. See the Saver class for more information on restoring saved variables.

Example of restoring the shadow variable values:

```
# Create a Saver that loads variables from their saved shadow
values.
shadow_var0_name = ema.average_name(var0)
shadow_var1_name = ema.average_name(var1)
saver = tf.train.Saver({shadow_var0_name: var0, shadow_var1_name:
var1})
saver.restore(...checkpoint filename...)
```

```
# var0 and var1 now hold the moving average values
```

`tf.train.ExponentialMovingAverage.__init__(decay, num_updates=None, name='ExponentialMovingAverage')`

Creates a new ExponentialMovingAverage object.

The `Apply()` method has to be called to create shadow variables and add ops to maintain moving averages.

The optional `num_updates` parameter allows one to tweak the decay rate dynamically. . It is typical to pass the count of training steps, usually kept in a variable that is incremented at each step, in which case the decay rate is lower at the start of training. This makes moving averages move faster. If passed, the actual decay rate used is:
`min(decay, (1 + num_updates) / (10 + num_updates))`

Args:

- `decay`: Float. The decay to use.
- `num_updates`: Optional count of number of updates applied to variables.
- `name`: String. Optional prefix name to use for the name of ops added in `Apply()`.

`tf.train.ExponentialMovingAverage.apply(var_list=None)`

Maintains moving averages of variables.

`var_list` must be a list of `Variable` or `Tensor` objects. This method creates shadow variables for all elements of `var_list`. Shadow variables for `Variable` objects are initialized to the variable's initial value. They will be added to the `GraphKeys.MOVING_AVERAGE_VARIABLES` collection. For `Tensor` objects, the shadow variables are initialized to 0.

shadow variables are created with `trainable=False` and added to the `GraphKeys.ALL_VARIABLES`collection. They will be returned by calls to `tf.all_variables()`.

Returns an op that updates all shadow variables as described above.

Note that `apply()` can be called multiple times with different lists of variables.

Args:

- `var_list`: A list of Variable or Tensor objects. The variables and Tensors must be of types float32 or float64.

Returns:

An Operation that updates the moving averages.

Raises:

- `TypeError`: If the arguments are not all float32 or float64.
- `ValueError`: If the moving average of one of the variables is already being computed.

`tf.train.ExponentialMovingAverage.average_name(var)`

Returns the name of the `Variable` holding the average for `var`.

The typical scenario for `ExponentialMovingAverage` is to compute moving averages of variables during training, and restore the variables from the computed moving averages during evaluations. To restore variables, you have to know the name of the shadow variables. That name and the original variable can then be passed to a `Saver()` object to restore the variable from the moving average value with: `saver = tf.train.Saver({ema.average_name(var): var})`

`average_name()` can be called whether or not `apply()` has been called.

Args:

- `var`: A `Variable` object.

Returns:

A string: The name of the variable that will be used or was used by the `ExponentialMovingAverage class`to hold the moving average of `var`.

`tf.train.ExponentialMovingAverage.average(var)`

Returns the `Variable` holding the average of `var`.

Args:

- `var`: A `Variable` object.

Returns:

A `Variable` object or `None` if the moving average of `var` is not maintained..

`tf.train.ExponentialMovingAverage.variables_to_restore()`

Returns a map of names to `Variables` to restore.

If a variable has a moving average, use the moving average variable name as the restore name; otherwise, use the variable name.

For example,

```
  variables_to_restore = ema.variables_to_restore()
  saver = tf.train.Saver(variables_to_restore)
```

Below is an example of such mapping:

```
  conv/batchnorm/gamma/ExponentialMovingAverage:
conv/batchnorm/gamma,
  conv_4/conv2d_params/ExponentialMovingAverage:
conv_4/conv2d_params,
  global_step: global_step
```

Returns:

A map from restore_names to variables. The restore_name can be the moving_average version of the variable name if it exist, or the original variable name.

Coordinator and QueueRunner

See Threading and Queues for how to use threads and queues. For documentation on the Queue API, see Queues.

`class tf.train.Coordinator`

A coordinator for threads.

This class implements a simple mechanism to coordinate the termination of a set of threads.

Usage:

```
# Create a coordinator.
coord = Coordinator()
# Start a number of threads, passing the coordinator to each of
them.
...start thread 1...(coord, ...)
...start thread N...(coord, ...)
# Wait for all the threads to terminate.
coord.join(threads)
```

Any of the threads can call `coord.request_stop()` to ask for all the threads to stop. To cooperate with the requests, each thread must check for `coord.should_stop()` on a regular

basis. `coord.should_stop()` returns `True` as soon

as `coord.request_stop()` has been called.

A typical thread running with a coordinator will do something like:

```
while not coord.should_stop():
  ...do some work...
```

Exception handling:

A thread can report an exception to the coordinator as part of the `should_stop()` call. The exception will be re-raised from the `coord.join()` call.

Thread code:

```
try:
  while not coord.should_stop():
    ...do some work...
except Exception as e:
  coord.request_stop(e)
```

Main code:

```
try:
  ...
  coord = Coordinator()
  # Start a number of threads, passing the coordinator to each of
them.
  ...start thread 1...(coord, ...)
  ...start thread N...(coord, ...)
  # Wait for all the threads to terminate.
  coord.join(threads)
except Exception as e:
  ...exception that was passed to coord.request_stop()
```

To simplify the thread implementation, the Coordinator provides a context handler `stop_on_exception()` that automatically requests a stop if an exception is raised. Using the context handler the thread code above can be written as:

```
with coord.stop_on_exception():
  while not coord.should_stop():
    ...do some work...
```

Grace period for stopping:

After a thread has called `coord.request_stop()` the other threads have a fixed time to stop, this is called the 'stop grace period' and defaults to 2 minutes. If any of the threads is still alive after the grace period expires`coord.join()` raises a RuntimeException reporting the laggards.

```
try:
  ...
  coord = Coordinator()
  # Start a number of threads, passing the coordinator to each of
them.
  ...start thread 1...(coord, ...)
  ...start thread N...(coord, ...)
  # Wait for all the threads to terminate, give them 10s grace
period
  coord.join(threads, stop_grace_period_secs=10)
except RuntimeException:
  ...one of the threads took more than 10s to stop after
request_stop()
  ...was called.
except Exception:
  ...exception that was passed to coord.request_stop()
```

`tf.train.Coordinator.__init__()`

Create a new Coordinator.

`tf.train.Coordinator.clear_stop()`

Clears the stop flag.

After this is called, calls to `should_stop()` will return `False`.

`tf.train.Coordinator.join(threads, stop_grace_period_secs=120)`

Wait for threads to terminate.

Blocks until all `threads` have terminated or `request_stop()` is called.
After the threads stop, if an `exc_info` was passed to `request_stop`, that exception is re-raised.
Grace period handling: When `request_stop()` is called, threads are given 'stop_grace_period_secs' seconds to terminate. If any of them is still alive after that period expires, a `RuntimeError` is raised. Note that if an`exc_info` was passed to `request_stop()` then it is raised instead of that `RuntimeError`.

Args:

- `threads`: List of `threading.Threads`. The started threads to join.
- `stop_grace_period_secs`: Number of seconds given to threads to stop after `request_stop()` has been called.

Raises:

- `RuntimeError`: If any thread is still alive after `request_stop()` is called and the grace period expires.

`tf.train.Coordinator.request_stop(ex=None)`

Request that the threads stop.

After this is called, calls to `should_stop()` will return `True`.

Args:

- `ex`: Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

`tf.train.Coordinator.should_stop()`

Check if stop was requested.

Returns:

True if a stop was requested.

`tf.train.Coordinator.stop_on_exception()`

Context manager to request stop when an Exception is raised.

Code that uses a coordinator must catch exceptions and pass them to the `request_stop()` method to stop the other threads managed by the coordinator.

This context handler simplifies the exception handling. Use it as follows:

```
with coord.stop_on_exception():
  # Any exception raised in the body of the with
  # clause is reported to the coordinator before terminating
  # the execution of the body.
  ...body...
```

This is completely equivalent to the slightly longer code:

```
try:
  ...body...
exception Exception as ex:
  coord.request_stop(ex)
```

Yields:

nothing.

`tf.train.Coordinator.wait_for_stop(timeout=None)`

Wait till the Coordinator is told to stop.

Args:

- `timeout`: Float. Sleep for up to that many seconds waiting for should_stop() to become True.

Returns:

True if the Coordinator is told stop, False if the timeout expired.

`class tf.train.QueueRunner`

Holds a list of enqueue operations for a queue, each to be run in a thread.

Queues are a convenient TensorFlow mechanism to compute tensors asynchronously using multiple threads. For example in the canonical 'Input Reader' setup one set of threads generates filenames in a queue; a second set of threads read records from the files, processes them, and enqueues tensors on a second queue; a third set of threads dequeues these input records to construct batches and runs them through training operations.

There are several delicate issues when running multiple threads that way: closing the queues in sequence as the input is exhausted, correctly catching and reporting exceptions, etc.

The `QueueRunner`, combined with the `Coordinator`, helps handle these issues.

`tf.train.QueueRunner.__init__(queue=None, enqueue_ops=None, close_op=None, cancel_op=None, queue_runner_def=None)`

Create a QueueRunner.

On construction the `QueueRunner` adds an op to close the queue. That op will be run if the enqueue ops raise exceptions. When you later call the `create_threads()` method,

the `QueueRunner` will create one thread for each op in`enqueue_ops`. Each thread will run its enqueue op in parallel with the other threads. The enqueue ops do not have to all be the same op, but it is expected that they all enqueue tensors in `queue`.

Args:

- `queue`: A `Queue`.
- `enqueue_ops`: List of enqueue ops to run in threads later.
- `close_op`: Op to close the queue. Pending enqueue ops are preserved.
- `cancel_op`: Op to close the queue and cancel pending enqueue ops.

* `queue_runner_def`: Optional `QueueRunnerDef` protocol buffer. If specified, recreates the QueueRunner from its contents. `queue_runner_def` and the other arguments are mutually exclusive.

Raises:

* `ValueError`: If both `queue_runner_def` and `queue` are both specified.
* `ValueError`: If `queue` or `enqueue_ops` are not provided when not restoring from `queue_runner_def`.

`tf.train.QueueRunner.cancel_op`

`tf.train.QueueRunner.close_op`

`tf.train.QueueRunner.create_threads(sess, coord=None, daemon=False, start=False)`

Create threads to run the enqueue ops.

This method requires a session in which the graph was launched. It creates a list of threads, optionally starting them. There is one thread for each op passed in `enqueue_ops`.

The `coord` argument is an optional coordinator, that the threads will use to terminate together and report exceptions. If a coordinator is given, this method starts an additional thread to close the queue when the coordinator requests a stop.

This method may be called again as long as all threads from a previous call have stopped.

Args:

- `sess`: A `Session`.
- `coord`: Optional `Coordinator` object for reporting errors and checking stop conditions.
- `daemon`: Boolean. If `True` make the threads daemon threads.
- `start`: Boolean. If `True` starts the threads. If `False` the caller must call the `start()` method of the returned threads.

Returns:

A list of threads.

Raises:

- `RuntimeError`: If threads from a previous call to `create_threads()` are still running.

`tf.train.QueueRunner.enqueue_ops`

`tf.train.QueueRunner.exceptions_raised`

Exceptions raised but not handled by the `QueueRunner` threads. Exceptions raised in queue runner threads are handled in one of two ways depending on whether or not a`Coordinator` was passed to `create_threads()`:

- With a `Coordinator`, exceptions are reported to the coordinator and forgotten by the `QueueRunner`.
- Without a `Coordinator`, exceptions are captured by the `QueueRunner` and made available in this`exceptions_raised` property.

Returns:

A list of Python `Exception` objects. The list is empty if no exception was captured. (No exceptions are captured when using a Coordinator.)

`tf.train.QueueRunner.from_proto(queue_runner_def)`

`tf.train.QueueRunner.name`

The string name of the underlying Queue.

`tf.train.QueueRunner.queue`

`tf.train.QueueRunner.to_proto()`

Converts this `QueueRunner` to a `QueueRunnerDef` protocol buffer.

Returns:

A `QueueRunnerDef` protocol buffer.

`tf.train.add_queue_runner(qr, collection='queue_runners')`

Adds a `QueueRunner` to a collection in the graph.

When building a complex model that uses many queues it is often difficult to gather all the queue runners that need to be run. This convenience function allows you to add a queue runner to a well known collection in the graph.

The companion method `start_queue_runners()` can be used to start threads for all the collected queue runners.

Args:

- `qr`: A `QueueRunner`.
- `collection`: A `GraphKey` specifying the graph collection to add the queue runner to. Defaults to`GraphKeys.QUEUE_RUNNERS`.

`tf.train.start_queue_runners(sess=None, coord=None, daemon=True, start=True, collection='queue_runners')`

Starts all queue runners collected in the graph.

This is a companion method to `add_queue_runner()`. It just starts threads for all queue runners collected in the graph. It returns the list of all threads.

Args:

- `sess`: `Session` used to run the queue ops. Defaults to the default session.
- `coord`: Optional `Coordinator` for coordinating the started threads.
- `daemon`: Whether the threads should be marked as `daemons`, meaning they don't block program exit.
- `start`: Set to `False` to only create the threads, not start them.
- `collection`: A `GraphKey` specifying the graph collection to get the queue runners from. Defaults to`GraphKeys.QUEUE_RUNNERS`.

Returns:

A list of threads.

Summary Operations

The following ops output `Summary` protocol buffers as serialized string tensors.
You can fetch the output of a summary op in a session, and pass it to a SummaryWriter to append it to an event file. Event files contain `Event` protos that can contain `Summary` protos along with the timestamp and step. You can then use TensorBoard to visualize the

contents of the event files. See TensorBoard and Summaries for more details.

```
tf.scalar_summary(tags, values, collections=None, name=None)
```

Outputs a `Summary` protocol buffer with scalar values.

The input `tags` and `values` must have the same shape. The generated summary has a summary value for each tag-value pair in `tags` and `values`.

Args:

- `tags`: A `string` `Tensor`. Tags for the summaries.
- `values`: A real numeric Tensor. Values for the summaries.
- `collections`: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name`: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

```
tf.image_summary(tag, tensor, max_images=3, collections=None, name=None)
```

Outputs a `Summary` protocol buffer with images.

The summary has up to `max_images` summary values containing images. The images are built from `tensor`which must be 4-D with shape `[batch_size, height, width, channels]` and where `channels` can be:

- 1: `tensor` is interpreted as Grayscale.
- 3: `tensor` is interpreted as RGB.
- 4: `tensor` is interpreted as RGBA.

The images have the same number of channels as the input tensor. For float input, the values are normalized one image at a time to fit in the range `[0, 255]`. `uint8` values are unchanged. The op uses two different normalization algorithms:

- If the input values are all positive, they are rescaled so the largest one is 255.
- If any input value is negative, the values are shifted so input value 0.0 is at 127. They are then rescaled so that either the smallest value is 0, or the largest one is 255.

The `tag` argument is a scalar `Tensor` of type `string`. It is used to build the `tag` of the summary values:

- If `max_images` is 1, the summary value tag is '*tag*/image'.
- If `max_images` is greater than 1, the summary value tags are generated sequentially as '*tag*/image/0', '*tag*/image/1', etc.

Args:

- `tag`: A scalar `Tensor` of type `string`. Used to build the `tag` of the summary values.
- `tensor`: A 4-D `uint8` or `float32` `Tensor` of shape `[batch_size, height, width, channels]` where `channels` is 1, 3, or 4.
- `max_images`: Max number of batch elements to generate images for.
- `collections`: Optional list of ops.GraphKeys. The collections to add the summary to. Defaults to [ops.GraphKeys.SUMMARIES]
- `name`: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

```
tf.histogram_summary(tag, values, collections=None,
name=None)
```

Outputs a `Summary` protocol buffer with a histogram.

The generated `Summary` has one summary value containing a histogram for `values`.

This op reports an `OutOfRange` error if any value is not finite.

Args:

- `tag`: A `string` `Tensor`. 0-D. Tag to use for the summary value.
- `values`: A real numeric `Tensor`. Any shape. Values to use to build the histogram.

- `collections`: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name`: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

`tf.nn.zero_fraction(value, name=None)`

Returns the fraction of zeros in `value`.

If `value` is empty, the result is `nan`.

This is useful in summaries to measure and report sparsity. For example,

```
z = tf.Relu(...)
summ = tf.scalar_summary('sparsity', tf.zero_fraction(z))
```

Args:

- `value`: A tensor of numeric type.
- `name`: A name for the operation (optional).

Returns:

The fraction of zeros in `value`, with type `float32`.

`tf.merge_summary(inputs, collections=None, name=None)`

Merges summaries.

This op creates a `Summary` protocol buffer that contains the union of all the values in the input summaries.
When the Op is run, it reports an `InvalidArgument` error if multiple values in the summaries to merge use the same tag.

Args:

- `inputs`: A list of `string` `Tensor` objects containing

  serialized `Summary` protocol buffers.
- `collections`: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name`: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer resulting from the merging.

`tf.merge_all_summaries(key='summaries')`

Merges all summaries collected in the default graph.

Args:

- `key`: `GraphKey` used to collect the summaries. Defaults to `GraphKeys.SUMMARIES`.

Returns:

If no summaries were collected, returns None. Otherwise returns a scalar `Tensor` of type`string` containing the serialized `Summary` protocol buffer resulting from the merging.

Adding Summaries to Event Files

See Summaries and TensorBoard for an overview of summaries, event files, and visualization in TensorBoard.

`class tf.train.SummaryWriter`

Writes `Summary` protocol buffers to event files.

The `SummaryWriter` class provides a mechanism to create an event file in a given directory and add summaries and events to it. The class updates the file contents asynchronously. This allows a training program to call methods to add data to the file directly from the training loop, without slowing down training.

`tf.train.SummaryWriter.__init__(logdir, graph_def=None, max_queue=10, flush_secs=120)`

Creates a `SummaryWriter` and an event file.

On construction the summary writer creates a new event file in `logdir`. This event file will contain `Event`protocol buffers constructed when you call one of the following functions: `add_summary()`,`add_session_log()`, `add_event()`,

or `add_graph()`.

If you pass a `graph_def` protocol buffer to the constructor it is added

to the event file. (This is equivalent to calling `add_graph()` later).

TensorBoard will pick the graph from the file and display it graphically so you can interactively explore the graph you built. You will usually pass the graph from the session in which you launched it:

```
...create a graph...
# Launch the graph in a session.
sess = tf.Session()
# Create a summary writer, add the 'graph_def' to the event file.
writer = tf.train.SummaryWriter(<some-directory>, sess.graph_def)
```

The other arguments to the constructor control the asynchronous writes to the event file:

- `flush_secs`: How often, in seconds, to flush the added summaries and events to disk.
- `max_queue`: Maximum number of summaries or events pending to be written to disk before one of the 'add' calls block.

Args:

- `logdir`: A string. Directory where event file will be written.
- `graph_def`: A `GraphDef` protocol buffer.
- `max_queue`: Integer. Size of the queue for pending events and summaries.

* `flush_secs`: Number. How often, in seconds, to flush the pending events and summaries to disk.

`tf.train.SummaryWriter.add_summary(summary, global_step=None)`

Adds a `Summary` protocol buffer to the event file.

This method wraps the provided summary in an `Event` protocol buffer and adds it to the event file.
You can pass the result of evaluating any summary op, using [`Session.run()`](client.md#Session.run] or`Tensor.eval()`, to this

function. Alternatively, you can pass a `tf.Summary` protocol buffer that you populate with your own data. The latter is commonly done to report evaluation results in event files.

Args:

* `summary`: A `Summary` protocol buffer, optionally serialized as a string.
* `global_step`: Number. Optional global step value to record with the summary.

`tf.train.SummaryWriter.add_session_log(session_log, global_step=None)`

Adds a `SessionLog` protocol buffer to the event file.

This method wraps the provided session in an `Event` procotol buffer and adds it to the event file.

Args:

- `session_log`: A `SessionLog` protocol buffer.
- `global_step`: Number. Optional global step value to record with the summary.

`tf.train.SummaryWriter.add_event(event)`

Adds an event to the event file.

Args:

- `event`: An `Event` protocol buffer.

`tf.train.SummaryWriter.add_graph(graph_def, global_step=None)`

Adds a `GraphDef` protocol buffer to the event file.

The graph described by the protocol buffer will be displayed by TensorBoard. Most users pass a graph in the constructor instead.

Args:

- `graph_def`: A `GraphDef` protocol buffer.
- `global_step`: Number. Optional global step counter to record with the graph.

`tf.train.SummaryWriter.flush()`

Flushes the event file to disk.

Call this method to make sure that all pending events have been written to disk.

`tf.train.SummaryWriter.close()`

Flushes the event file to disk and close the file.

Call this method when you do not need the summary writer anymore.

`tf.train.summary_iterator(path)`

An iterator for reading `Event` protocol buffers from an event file.

You can use this function to read events written to an event file. It returns a Python iterator that yields `Event`protocol buffers.

Example: Print the contents of an events file.

```
for e in tf.train.summary_iterator(path to events file):
    print(e)
```

Example: Print selected summary values.

```
# This example supposes that the events file contains summaries
with a
# summary value tag 'loss'.  These could have been added by
calling
# `add_summary()`, passing the output of a scalar summary op
created with
# with: `tf.scalar_summary(['loss'], loss_tensor)`.
for e in tf.train.summary_iterator(path to events file):
    for v in e.summary.value:
        if v.tag == 'loss':
```

```
            print(v.simple_value)
```

See the protocol buffer definitions of Event and Summary for more information about their attributes.

Args:

- `path`: The path to an event file created by a `SummaryWriter`.

Yields:

`Event` protocol buffers.

Training utilities

`tf.train.global_step(sess, global_step_tensor)`

Small helper to get the global step.

```
# Creates a variable to hold the global_step.
global_step_tensor = tf.Variable(10, trainable=False,
name='global_step')
# Creates a session.
sess = tf.Session()
# Initializes the variable.
sess.run(global_step_tensor.initializer)
print('global_step: %s' % tf.train.global_step(sess,
global_step_tensor))

global_step: 10
```

Args:

- `sess`: A brain `Session` object.

* `global_step_tensor`: `Tensor` or the `name` of the operation that contains the global step.

Returns:

The global step value.

`tf.train.write_graph(graph_def, logdir, name, as_text=True)`

Writes a graph proto on disk.

The graph is written as a binary proto unless `as_text` is `True`.

```
v = tf.Variable(0, name='my_variable')
sess = tf.Session()
tf.train.write_graph(sess.graph_def, '/tmp/my-model',
'train.pbtxt')
```

Args:

* `graph_def`: A `GraphDef` protocol buffer.
* `logdir`: Directory where to write the graph.
* `name`: Filename for the graph.
* `as_text`: If `True`, writes the graph as an ASCII proto.

Other Functions and Classes

`class tf.train.LooperThread`

A thread that runs code repeatedly, optionally on a timer.

This thread class is intended to be used with a `Coordinator`. It repeatedly runs code specified either as `target`and `args` or by the `run_loop()` method.

Before each run the thread checks if the coordinator has requested stop. In that case the looper thread terminates immediately.

If the code being run raises an exception, that exception is reported to the coordinator and the thread terminates. The coordinator will then request all the other threads it coordinates to stop.

You typically pass looper threads to the supervisor `Join()` method.

`tf.train.LooperThread.__init__(coord, timer_interval_secs, target=None, args=None)`

Create a LooperThread.

Args:

- `coord`: A Coordinator.
- `timer_interval_secs`: Time boundaries at which to call Run(), or None if it should be called back to back.
- `target`: Optional callable object that will be executed in the thread.
- `args`: Optional arguments to pass to `target` when calling it.

Raises:

- `ValueError`: If one of the arguments is invalid.

`tf.train.LooperThread.daemon`

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

`tf.train.LooperThread.getName()`

`tf.train.LooperThread.ident`

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

`tf.train.LooperThread.isAlive()`

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

`tf.train.LooperThread.isDaemon()`

`tf.train.LooperThread.is_alive()`

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

`tf.train.LooperThread.join(timeout=None)`

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates -- either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened -- if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

`tf.train.LooperThread.loop(coord, timer_interval_secs, target, args=None)`

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is None the thread

calls `target(args)` repeatedly. Otherwise `target(args)` is called

every `timer_interval_secs` seconds. The thread terminates when a stop of the coordinator is requested.

Args:

- `coord`: A Coordinator.
- `timer_interval_secs`: Number. Time boundaries at which to

  call `target`.
- `target`: A callable object.
- `args`: Optional arguments to pass to `target` when calling it.

Returns:

The started thread.

`tf.train.LooperThread.name`

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

`tf.train.LooperThread.run()`

`tf.train.LooperThread.run_loop()`

Called at 'timer_interval_secs' boundaries.

`tf.train.LooperThread.setDaemon(daemonic)`

`tf.train.LooperThread.setName(name)`

`tf.train.LooperThread.start()`

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

`tf.train.LooperThread.start_loop()`

Called when the thread starts.

`tf.train.export_meta_graph(filename=None, meta_info_def=None, graph_def=None, saver_def=None, collection_list=None, as_text=False)`

Returns `MetaGraphDef` proto. Optionally writes it to filename.

This function exports the graph, saver, and collection objects into `MetaGraphDef` protocol buffer with the intension of it being imported at a later time or location to restart training, run inference, or be a subgraph.

Args:

- `filename`: Optional filename including the path for writing the generated `MetaGraphDef` protocol buffer.
- `meta_info_def`: `MetaInfoDef` protocol buffer.
- `graph_def`: `GraphDef` protocol buffer.
- `saver_def`: `SaverDef` protocol buffer.
- `collection_list`: List of string keys to collect.
- `as_text`: If `True`, writes the `MetaGraphDef` as an ASCII proto.

Returns:

A `MetaGraphDef` proto.

`tf.train.generate_checkpoint_state_proto(save_dir, model_checkpoint_path, all_model_checkpoint_paths=None)`

Generates a checkpoint state proto.

Args:

- `save_dir`: Directory where the model was saved.
- `model_checkpoint_path`: The checkpoint file.
- `all_model_checkpoint_paths`: List of strings. Paths to all not-yet-deleted checkpoints, sorted from oldest to newest. If this is a non-empty list, the last element must be equal to model_checkpoint_path. These paths are also saved in the CheckpointState proto.

Returns:

CheckpointState proto with model_checkpoint_path and all_model_checkpoint_paths updated to either absolute paths or relative paths to the current save_dir.

`tf.train.import_meta_graph(meta_graph_or_file)`

Recreates a Graph saved in a `MetaGraphDef` proto.

This function reads from a file containing a `MetaGraphDef` proto, adds all the nodes from the graph_def proto to the current graph, recreates all the collections, and returns a saver from saver_def.

In combination with `export_meta_graph()`, this function can be used to

- Serialize a graph along with other Python objects such as `QueueRunner`, `Variable` into a`MetaGraphDef`.
- Restart training from a saved graph and checkpoints.
- Run inference from a saved graph and checkpoints.

Args:

- `meta_graph_or_file`: `MetaGraphDef` protocol buffer or filename (including the path) containing a`MetaGraphDef`.

Returns:

A saver constructed rom `saver_def` in `MetaGraphDef`.

Wraps python functions

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

Contents

Script Language Operators.

TensorFlow provides allows you to wrap python/numpy functions as TensorFlow operators.

Other Functions and Classes

`tf.py_func(func, inp, Tout, name=None)`

Wraps a python function and uses it as a tensorflow op.

Given a python function `func`, which takes numpy arrays as its inputs and returns numpy arrays as its outputs. E.g.,

def my_func(x): return np.sinh(x) inp = tf.placeholder(..., tf.float32) y = py_func(my_func, [inp], [tf.float32])

The above snippet constructs a tf graph which invokes a numpy sinh(x) as an op in the graph.

Args:

- `func`: A python function.
- `inp`: A list of `Tensor`.
- `Tout`: A list of tensorflow data types indicating what `func` returns.
- `name`: A name for the operation (optional).

Returns:

A list of `Tensor` which `func` computes.

Testing

Contents

Unit tests

TensorFlow provides a convenience class inheriting from `unittest.TestCase` which adds methods relevant to TensorFlow tests. Here is an example:

```
import tensorflow as tf


class SquareTest(tf.test.TestCase):

  def testSquare(self):
    with self.test_session():
      x = tf.square([2, 3])
      self.assertAllEqual(x.eval(), [4, 9])


if __name__ == '__main__':
  tf.test.main()
```

`tf.test.TestCase` inherits from `unittest.TestCase` but adds a few additional methods. We will document these methods soon.

`tf.test.main()`

Runs all unit tests.

Utilities

`tf.test.assert_equal_graph_def(actual, expected)`

Asserts that two `GraphDef`s are (mostly) the same.

Compares two `GraphDef` protos for equality, ignoring versions and ordering of nodes, attrs, and control inputs. Node names are used to match up nodes between the graphs, so the naming of nodes must be consistent.

Args:

- `actual`: The `GraphDef` we have.
- `expected`: The `GraphDef` we expected.

Raises:

- `AssertionError`: If the `GraphDef`s do not match.
- `TypeError`: If either argument is not a `GraphDef`.

`tf.test.get_temp_dir()`

Returns a temporary directory for use during tests.

There is no need to delete the directory after the test.

Returns:

The temporary directory.

`tf.test.is_built_with_cuda()`

Returns whether TensorFlow was built with CUDA (GPU) support.

Gradient checking

`compute_gradient` and `compute_gradient_error` perform numerical differentiation of graphs for comparison against registered analytic gradients.

`tf.test.compute_gradient(x, x_shape, y, y_shape, x_init_value=None, delta=0.001, init_targets=None)`

Computes and returns the theoretical and numerical Jacobian.

Args:

- `x`: a tensor or list of tensors
- `x_shape`: the dimensions of x as a tuple or an array of ints. If x is a list, then this is the list of shapes.
- `y`: a tensor
- `y_shape`: the dimensions of y as a tuple or an array of ints.
- `x_init_value`: (optional) a numpy array of the same shape as "x" representing the initial value of x. If x is a list, this should be a list of

numpy arrays. If this is none, the function will pick a random tensor as the initial value.

- `delta`: (optional) the amount of perturbation.
- `init_targets`: list of targets to run to initialize model params.

  TODO(mrry): remove this argument.

Returns:

Two 2-d numpy arrays representing the theoretical and numerical Jacobian for dy/dx. Each has "x_size" rows and "y_size" columns where "x_size" is the number of elements in x and "y_size" is the number of elements in y. If x is a list, returns a list of two numpy arrays.

`tf.test.compute_gradient_error(x, x_shape, y, y_shape, x_init_value=None, delta=0.001, init_targets=None)`

Computes the gradient error.

Computes the maximum error for dy/dx between the computed Jacobian and the numerically estimated Jacobian.

This function will modify the tensors passed in as it adds more operations and hence changing the consumers of the operations of the input tensors.

This function adds operations to the current session. To compute the error using a particular device, such as a GPU, use the standard methods for setting a device (e.g. using with sess.graph.device() or setting a device function in the session constructor).

Args:

- `x`: a tensor or list of tensors

- `x_shape`: the dimensions of x as a tuple or an array of ints. If x is a list, then this is the list of shapes.
- `y`: a tensor
- `y_shape`: the dimensions of y as a tuple or an array of ints.
- `x_init_value`: (optional) a numpy array of the same shape as "x" representing the initial value of x. If x is a list, this should be a list of numpy arrays. If this is none, the function will pick a random tensor as the initial value.
- `delta`: (optional) the amount of perturbation.
- `init_targets`: list of targets to run to initialize model params. TODO(mrry): Remove this argument.

Returns:

The maximum error in between the two Jacobians.

Layers (contrib)

Contents

- Regularizers
- `tf.contrib.layers.l1_regularizer(scale)`
- `tf.contrib.layers.l2_regularizer(scale)`
- Initializers
- `tf.contrib.layers.xavier_initializer(uniform=True, seed=None, dtype=tf.float32)`
- `tf.contrib.layers.xavier_initializer_conv2d(uniform=True, seed=None, dtype=tf.float32)`
- Summaries
- `tf.contrib.layers.summarize_activation(op)`
- `tf.contrib.layers.summarize_tensor(tensor)`
- `tf.contrib.layers.summarize_tensors(tensors, summarizer=summarize_tensor)`
- `tf.contrib.layers.summarize_collection(collection, name_filter=None, summarizer=summarize_tensor)`
- `tf.contrib.layers.summarize_activations(name_filter=None, summarizer=summarize_activation)`
- Other Functions and Classes
- `tf.contrib.layers.assert_same_float_dtype(tensors=None, dtype=None)`

Ops for building neural network layers, regularizers, summaries, etc.

Higher level ops for building neural network layers.

This package provides several ops that take care of creating variables that are used internally in a consistent way and provide the building blocks for many common machine learning algorithms.

```
tf.contrib.layers.convolution2d(x, num_output_channels,
kernel_size, activation_fn=None, stride=(1, 1),
padding='SAME', weight_init=_initializer,
bias_init=_initializer, name=None,
weight_collections=None, bias_collections=None,
output_collections=None, weight_regularizer=None,
bias_regularizer=None)
```

Adds the parameters for a conv2d layer and returns the output.

A neural network convolution layer is generally defined as: $y=f(conv2d(w,x)+b)$y=f(conv2d(w,x)+b) where f is given by`activation_fn`, conv2d is `tf.nn.conv2d` and `x` has shape `[batch, height, width, channels]`. The output of this op is of shape `[batch, out_height, out_width, num_output_channels]`, where`out_width` and `out_height` are determined by the `padding` argument. See `conv2D` for details.

This op creates `w` and optionally `b` and adds various summaries that can be useful for visualizing learning or diagnosing training problems. Bias can be disabled by setting `bias_init` to `None`.

The variable creation is compatible with `tf.variable_scope` and so can be reused with`tf.variable_scope` or `tf.make_template`.
Most of the details of variable creation can be controlled by specifying the initializers (`weight_init` and`bias_init`) and which collections to place the created variables in (`weight_collections` and`bias_collections`).
A per layer regularization can be specified by setting `weight_regularizer`. This is only applied to weights and not the bias.

Args:

- `x`: A 4-D input `Tensor`.
- `num_output_channels`: The number of output channels (i.e. the size of the last dimension of the output).
- `kernel_size`: A length 2 `list` or `tuple` containing the kernel size.

- `activation_fn`: A function that requires a single Tensor that is applied as a non-linearity.
- `stride`: A length 2 `list` or `tuple` specifying the stride of the sliding window across the image.
- `padding`: A `string` from: "SAME", "VALID". The type of padding algorithm to use.
- `weight_init`: An optional initialization. If not specified, uses Xavier initialization (see`tf.learn.xavier_initializer`).
- `bias_init`: An initializer for the bias, defaults to 0. Set to`None` in order to disable bias.
- `name`: The name for this operation is used to name operations and to find variables. If specified it must be unique for this scope, otherwise a unique name starting with "convolution2d" will be created. See`tf.variable_op_scope` for details.
- `weight_collections`: List of graph collections to which weights are added.
- `bias_collections`: List of graph collections to which biases are added.
- `output_collections`: List of graph collections to which outputs are added.
- `weight_regularizer`: A regularizer like the result of `l1_regularizer` or `l2_regularizer`. Used for weights.
- `bias_regularizer`: A regularizer like the result of `l1_regularizer` or `l2_regularizer`. Used for biases.

Returns:

The result of applying a 2-D convolutional layer.

Raises:

- `ValueError`: If `kernel_size` or `stride` are not length 2.

```
tf.contrib.layers.fully_connected(x, num_output_units,
activation_fn=None, weight_init=_initializer,
bias_init=_initializer, name=None,
weight_collections=('weights',),
bias_collections=('biases',),
output_collections=('activations',),
weight_regularizer=None, bias_regularizer=None)
```

Adds the parameters for a fully connected layer and returns the output.

A fully connected layer is generally defined as a matrix multiply: `y = f(w * x + b)` where `f` is given by`activation_fn`.

If `activation_fn` is `None`, the result of `y = w * x + b` is returned.

This op creates `w` and optionally `b`. Bias (`b`) can be disabled by setting `bias_init` to `None`.

The variable creation is compatible with `tf.variable_scope` and so can be reused with`tf.variable_scope` or `tf.make_template`. Most of the details of variable creation can be controlled by specifying the initializers (`weight_init` and`bias_init`) and which in collections to place the created variables (`weight_collections` and`bias_collections`; note that the variables are always added to the `VARIABLES` collection). The output of the layer can be placed in custom collections using `output_collections`. The collections arguments default to`WEIGHTS`, `BIASES` and `ACTIVATIONS`, respectively.

A per layer regularization can be specified by setting `weight_regularizer` and `bias_regularizer`, which are applied to the weights and biases respectively, and whose output is added to the `REGULARIZATION_LOSSES`collection.

Args:

- `x`: The input `Tensor`.
- `num_output_units`: The size of the output.
- `activation_fn`: A function that requires a single Tensor that is applied as a non-linearity. If None is used, do not apply any activation.
- `weight_init`: An optional weight initialization, defaults to `xavier_initializer`.
- `bias_init`: An initializer for the bias, defaults to 0. Set to `None` in order to disable bias.
- `name`: The name for this operation is used to name operations and to find variables. If specified it must be unique for this scope, otherwise a unique name starting with "fully_connected" will be created. See`tf.variable_op_scope` for details.
- `weight_collections`: List of graph collections to which weights are added.
- `bias_collections`: List of graph collections to which biases are added.
- `output_collections`: List of graph collections to which outputs are added.

- `weight_regularizer`: A regularizer like the result of `l1_regularizer` or `l2_regularizer`. Used for weights.
- `bias_regularizer`: A regularizer like the result of `l1_regularizer` or `l2_regularizer`. Used for biases.

Returns:

The output of the fully connected layer.

Aliases for fully_connected which set a default activation function are available: `relu`, `relu6` and `linear`.

Regularizers

Regularization can help prevent overfitting. These have the signature `fn(weights)`. The loss is typically added to `tf.GraphKeys.REGULARIZATION_LOSS`

`tf.contrib.layers.l1_regularizer(scale)`

Returns a function that can be used to apply L1 regularization to weights.

L1 regularization encourages sparsity.

Args:

- `scale`: A scalar multiplier `Tensor`. 0.0 disables the regularizer.

Returns:

A function with signature `l1(weights, name=None)` that apply L1 regularization.

Raises:

- `ValueError`: If scale is outside of the range [0.0, 1.0] or if scale is not a float.

`tf.contrib.layers.l2_regularizer(scale)`

Returns a function that can be used to apply L2 regularization to weights.

Small values of L2 can help prevent overfitting the training data.

Args:

- `scale`: A scalar multiplier `Tensor`. 0.0 disables the regularizer.

Returns:

A function with signature `l2(weights, name=None)` that applies L2 regularization.

Raises:

- `ValueError`: If scale is outside of the range [0.0, 1.0] or if scale is not a float.

Initializers

Initializers are used to initialize variables with sensible values given their size, data type, and purpose.

`tf.contrib.layers.xavier_initializer(uniform=True, seed=None, dtype=tf.float32)`

Returns an initializer performing "Xavier" initialization for weights.

This function implements the weight initialization from:

Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

This initializer is designed to keep the scale of the gradients roughly the same in all layers. In uniform distribution this ends up being the range: `x = sqrt(6. / (in + out)); [-x, x]` and for normal distribution a standard deviation of `sqrt(3. / (in + out))` is used. The returned initializer assumes that the shape of the weight matrix to be initialized is `[in, out]`.

Args:

- `uniform`: Whether to use uniform or normal distributed random initialization.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer for a 2-D weight matrix.

Raises:

- `TypeError`: If dtype is not a floating point type.

`tf.contrib.layers.xavier_initializer_conv2d(uniform=True, seed=None, dtype=tf.float32)`

Returns an "Xavier" initializer for 2D convolution weights.

For details on the initialization performed, see `xavier_initializer`.

This function initializes a convolution weight variable which is assumed to be 4-D. The first two dimensions are expected to be the kernel size, the third dimension is the number of input channels, and the last dimension is the number of output channels.

The number of inputs is therefore `shape[0]*shape[1]*shape[2]`,

and the number of outputs is`shape[0]*shape[1]*shape[3]`.

Args:

- `uniform`: Whether to use uniform or normal distributed random initialization.
- `seed`: A Python integer. Used to create random seeds.

  See `set_random_seed` for behavior.
- `dtype`: The data type. Only floating point types are supported.

Returns:

An initializer for a 4-D weight matrix.

Raises:

- `TypeError`: If dtype is not a floating point type.

Summaries

Helper functions to summarize specific variables or ops.

`tf.contrib.layers.summarize_activation(op)`

Summarize an activation.

This applies the given activation and adds useful summaries specific to the activation.

Args:

- `op`: The tensor to summarize (assumed to be a layer activation).

Returns:

The summary op created to summarize `op`.

`tf.contrib.layers.summarize_tensor(tensor)`

Summarize a tensor using a suitable summary type.

This function adds a summary op for `tensor`. The type of summary depends on the shape of `tensor`. For scalars, a `scalar_summary` is created, for all other tensors, `histogram_summary` is used.

Args:

- `tensor`: The tensor to summarize

Returns:

The summary op created.

```
tf.contrib.layers.summarize_tensors(tensors,
summarizer=summarize_tensor)
```

Summarize a set of tensors.

```
tf.contrib.layers.summarize_collection(collection,
name_filter=None, summarizer=summarize_tensor)
```

Summarize a graph collection of tensors, possibly filtered by name.

The layers module defines convenience functions `summarize_variables`, `summarize_weights` and`summarize_biases`, which set the `collection` argument of `summarize_collection` to `VARIABLES`,`WEIGHTS` and `BIASES`, respectively.

`tf.contrib.layers.summarize_activations(name_filter=None, summarizer=summarize_activation)`

Summarize activations, using `summarize_activation` to summarize.

Other Functions and Classes

`tf.contrib.layers.assert_same_float_dtype(tensors=None, dtype=None)`

Validate and return float type based on `tensors` and `dtype`.

For ops such as matrix multiplication, inputs and weights must be of the same float type. This function validates that all `tensors` are the same type, validates that type is `dtype` (if supplied), and returns the type. Type must be`dtypes.float32` or `dtypes.float64`. If neither `tensors` nor `dtype` is supplied, default to`dtypes.float32`.

Args:

- `tensors`: Tensors of input values. Can include `None` elements, which will be ignored.
- `dtype`: Expected type.

Returns:

Validated type.

Raises:

- `ValueError`: if neither `tensors` nor `dtype` is supplied, or result is not float.

Utilities (contrib)

Contents

Utilities for dealing with Tensors.

Miscellaneous Utility Functions

`tf.contrib.util.constant_value(tensor)`

Returns the constant value of the given tensor, if efficiently calculable.

This function attempts to partially evaluate the given tensor, and returns its value as a numpy ndarray if this succeeds.

TODO(mrry): Consider whether this function should use a registration mechanism like gradients and ShapeFunctions, so that it is easily extensible.

Args:

- `tensor`: The Tensor to be evaluated.

Returns:

A numpy ndarray containing the constant value of the given `tensor`, or None if it cannot be calculated.

Raises:

- `TypeError`: if tensor is not an ops.Tensor.

```
tf.contrib.util.make_tensor_proto(values, dtype=None, shape=None)
```

Create a TensorProto.

Args:

- `values`: Values to put in the TensorProto.
- `dtype`: Optional tensor_pb2 DataType value.
- `shape`: List of integers representing the dimensions of tensor.

Returns:

A TensorProto. Depending on the type, it may contain data in the "tensor_content" attribute, which is not directly useful to Python programs. To access the values you should convert the proto back to a numpy ndarray with tensor_util.MakeNdarray(proto).

Raises:

- `TypeError`: if unsupported types are provided.

- `ValueError`: if arguments have inappropriate values.

make_tensor_proto accepts "values" of a python scalar, a python list, a numpy ndarray, or a numpy scalar.

If "values" is a python scalar or a python list, make_tensor_proto first convert it to numpy ndarray. If dtype is None, the conversion tries its best to infer the right numpy data type. Otherwise, the resulting numpy array has a compatible data type with the given dtype.

In either case above, the numpy ndarray (either the caller provided or the auto converted) must have the compatible type with dtype.

make_tensor_proto then converts the numpy array to a tensor proto.

If "shape" is None, the resulting tensor proto represents the numpy array precisely.

Otherwise, "shape" specifies the tensor's shape and the numpy array can not have more elements than what "shape" specifies.