Module: tf.compat.v1.debugging

- **Contents**
- Functions

Public API for tf.debugging namespace.

Functions

`Assert(...)`: Asserts that the given condition is true.
`assert_all_finite(...)`: Assert that the tensor does not contain any NaN's or Inf's.
`assert_equal(...)`: Assert the condition `x == y` holds element-wise.
`assert_greater(...)`: Assert the condition `x > y` holds element-wise.
`assert_greater_equal(...)`: Assert the condition `x >= y` holds element-wise.
`assert_integer(...)`: Assert that `x` is of integer dtype.
`assert_less(...)`: Assert the condition `x < y` holds element-wise.
`assert_less_equal(...)`: Assert the condition `x <= y` holds element-wise.
`assert_near(...)`: Assert the condition `x` and `y` are close element-wise.
`assert_negative(...)`: Assert the condition `x < 0` holds element-wise.
`assert_non_negative(...)`: Assert the condition `x >= 0` holds element-wise.
`assert_non_positive(...)`: Assert the condition `x <= 0` holds element-wise.
`assert_none_equal(...)`: Assert the condition `x != y` holds for all elements.
`assert_positive(...)`: Assert the condition `x > 0` holds element-wise.
`assert_proper_iterable(...)`: Static assert that values is a "proper" iterable.
`assert_rank(...)`: Assert `x` has rank equal to `rank`.
`assert_rank_at_least(...)`: Assert `x` has rank equal to `rank` or higher.
`assert_rank_in(...)`: Assert `x` has rank in `ranks`.
`assert_same_float_dtype(...)`: Validate and return float type based on `tensors` and `dtype`.
`assert_scalar(...)`: Asserts that the given `tensor` is a scalar (i.e. zero-dimensional).
`assert_shapes(...)`: Assert tensor shapes and dimension size relationships between tensors.
`assert_type(...)`: Statically asserts that the given `Tensor` is of the specified type.
`check_numerics(...)`: Checks a tensor for NaN and Inf values.
`get_log_device_placement(...)`: Get if device placements are logged.
`is_finite(...)`: Returns which elements of x are finite.
`is_inf(...)`: Returns which elements of x are Inf.
`is_nan(...)`: Returns which elements of x are NaN.
`is_non_decreasing(...)`: Returns `True` if `x` is non-decreasing.
`is_numeric_tensor(...)`: Returns `True` if the elements of `tensor` are numbers.
`is_strictly_increasing(...)`: Returns `True` if `x` is strictly increasing.
`set_log_device_placement(...)`: Set if device placements should be logged.

tf.compat.v1.debugging.assert_shapes

Assert tensor shapes and dimension size relationships between tensors.

```
tf.compat.v1.debugging.assert_shapes(
    shapes,
    data=None,
    summarize=None,
    message=None,
    name=None
)
```

Defined in `python/ops/check_ops.py`.

This Op checks that a collection of tensors shape relationships satisfies given constraints.

Example:

```
tf.assert_shapes({
  x: ('N', 'Q'),
  y: ('N', 'D'),
  param: ('Q',),
  scalar: ()
})
```

Example of adding a dependency to an operation:

```
with tf.control_dependencies([tf.assert_shapes(shapes)]):
  output = tf.matmul(x, y, transpose_a=True)
```

If `x`, `y`, `param` or `scalar` does not have a shape that satisfies all specified constraints, `message`, as well as the first `summarize` entries of the first encountered violating tensor are printed, and `InvalidArgumentError` is raised.
Size entries in the specified shapes are checked against other entries by their **hash**, except: - a size entry is interpreted as an explicit size if it can be parsed as an integer primitive. - a size entry is interpreted as *any* size if it is None or '.'.
If the first entry of a shape is `...` (type `Ellipsis`) or '*' that indicates a variable number of outer dimensions of unspecified size, i.e. the constraint applies to the inner-most dimensions only.
Scalar tensors and specified shapes of length zero (excluding the 'inner-most' prefix) are both treated as having a single dimension of size one.

Args:

- **`shapes`**: dictionary with (`Tensor` to shape) items. A shape must be an iterable.
- **`data`**: The tensors to print out if the condition is False. Defaults to error message and first few entries of the violating tensor.
- **`summarize`**: Print this many entries of the tensor.
- **`message`**: A string to prefix to the default message.
- **`name`**: A name for this operation (optional). Defaults to "assert_shapes".

Returns:

Op raising `InvalidArgumentError` unless all shape constraints are satisfied. If static checks determine all constraints are satisfied, a `no_op` is returned.

Raises:

- **`ValueError`**: If static checks determine any shape constraint is violated.

Module: tf.compat.v1.lite & tf.lite

- **Contents**
- Modules
- Classes
- Functions

Public API for tf.lite namespace.

Modules

`constants` module: Public API for tf.lite.constants namespace.
`experimental` module: Public API for tf.lite.experimental namespace.

Classes

`class Interpreter`: Interpreter interface for TensorFlow Lite Models.
`class OpHint`: A class that helps build tflite function invocations.
`class OpsSet`: Enum class defining the sets of ops available to generate TFLite models.
`class Optimize`: Enum defining the optimizations to apply when generating tflite graphs.
`class RepresentativeDataset`: Representative dataset to evaluate optimizations.
`class TFLiteConverter`: Convert a TensorFlow model into `output_format`.
`class TargetSpec`: Specification of target device.
`class TocoConverter`: Convert a TensorFlow model into `output_format` using TOCO.

Functions

`toco_convert(...)`: Convert a model using TOCO. (deprecated)

tf.compat.v1.lite.OpHint

- Contents
- Class OpHint
- __init__
- Child Classes
- Methods

Class `OpHint`

A class that helps build tflite function invocations.
Defined in `lite/python/op_hint.py`.
It allows you to take a bunch of TensorFlow ops and annotate the construction such that toco knows how to convert it to tflite. This embeds a pseudo function in a TensorFlow graph. This allows embedding high-level API usage information in a lower level TensorFlow implementation so that an alternative implementation can be substituted later.
Essentially, any "input" into this pseudo op is fed into an identity, and attributes are added to that input before being used by the constituent ops that make up the pseudo op. A similar process is done to any output that is to be exported from the current op.

`__init__`

```
__init__(

    function_name,

    level=1,

    children_inputs_mappings=None,

    **kwargs

)
```

Create a OpHint.

Args:

- **`function_name`**: Name of the function (the custom op name in tflite)
- **`level`**: OpHint level.
- **`children_inputs_mappings`**: Children OpHint inputs/outputs mapping. children_inputs_mappings should like below: "parent_first_child_input": [{"parent_input_index": num, "child_input_index": num},

...] "parent_last_child_output": [{"parent_output_index": num, "child_output_index": num}, ...] "internal_children_input_output": [{"child_input_index": num, "child_output_index": num}, ...]

- `**kwargs`: Keyword arguments of any constant attributes for the function.

Child Classes

`class OpHintArgumentTracker`

Methods

`add_input`

```
add_input(

    *args,

    **kwargs

)
```

Add a wrapped input argument to the hint.

Args:

- `*args`: The input tensor.
- `**kwargs`: "name" label "tag" a tag to group multiple arguments that will be aggregated. I.e. a string like 'cool_input'. Basically multiple inputs can be added to the same hint for parallel operations that will eventually be combined. An example would be static_rnn which creates multiple copies of state or inputs. "aggregate" aggregation strategy that is valid only for tag non None. Acceptable values are OpHint.AGGREGATE_FIRST, OpHint.AGGREGATE_LAST, and OpHint.AGGREGATE_STACK. "index_override" The global index to use. This corresponds to the argument order in the final stub that will be generated.

Returns:
The wrapped input tensor.

`add_inputs`

```
add_inputs(

    *args,

    **kwargs

)
```

Add a sequence of inputs to the function invocation.

Args:

- `*args`: List of inputs to be converted (should be Tf.Tensor).
- `**kwargs`: This allows 'names' which should be a list of names.

Returns:
Wrapped inputs (identity standins that have additional metadata). These are also are also tf.Tensor's.

add_output

```
add_output(

    *args,

    **kwargs

)
```

Add a wrapped output argument to the hint.

Args:

- **`*args`**: The output tensor.
- **`**kwargs`**: "name" label "tag" a tag to group multiple arguments that will be aggregated. I.e. a string like 'cool_input'. Basically multiple inputs can be added to the same hint for parallel operations that will eventually be combined. An example would be static_rnn which creates multiple copies of state or inputs. "aggregate" aggregation strategy that is valid only for tag non None. Acceptable values are OpHint.AGGREGATE_FIRST, OpHint.AGGREGATE_LAST, and OpHint.AGGREGATE_STACK. "index_override" The global index to use. This corresponds to the argument order in the final stub that will be generated.

Returns:
The wrapped output tensor.

add_outputs

```
add_outputs(

    *args,

    **kwargs

)
```

Add a sequence of outputs to the function invocation.

Args:

- **`*args`**: List of outputs to be converted (should be tf.Tensor).
- **`**kwargs`**: See

Returns:
Wrapped outputs (identity standins that have additional metadata). These are also tf.Tensor's.

Class Members

- `AGGREGATE_FIRST = 'first'`
- `AGGREGATE_LAST = 'last'`
- `AGGREGATE_STACK = 'stack'`
- `CHILDREN_INPUTS_MAPPINGS = '_tflite_children_ophint_inputs_mapping'`
- `FUNCTION_AGGREGATE_ATTR = '_tflite_function_aggregate'`
- `FUNCTION_INPUT_INDEX_ATTR = '_tflite_function_input_index'`
- `FUNCTION_LEVEL_ATTR = '_tflite_ophint_level'`
- `FUNCTION_NAME_ATTR = '_tflite_function_name'`

- `FUNCTION_OUTPUT_INDEX_ATTR = '_tflite_function_output_index'`
- `FUNCTION_SORT_INDEX_ATTR = '_tflite_function_sort_index'`
- `FUNCTION_UUID_ATTR = '_tflite_function_uuid'`
- `TFLITE_INPUT_INDICES = '_tflite_input_indices'`

tf.compat.v1.lite.OpHint.OpHintArgumentTracker

Class `OpHintArgumentTracker`

Conceptually tracks indices of arguments of "OpHint functions".

Defined in `lite/python/op_hint.py`.

The inputs and arguments of these functions both use an instance of the class so they can have independent numbering.

`__init__`

```
__init__(

    function_name,

    unique_function_id,

    node_name_prefix,

    attr_name,

    level=1,

    children_inputs_mappings=None

)
```

Initialize ophint argument.

Args:

- **`function_name`**: Name of the function that this tracks arguments for.
- **`unique_function_id`**: UUID of function that this tracks arguments for.
- **`node_name_prefix`**: How identities that are created are named.
- **`attr_name`**: Name of attribute to use to store the index for this hint. i.e. FUNCTION_INPUT_INDEX or FUNCTION_OUTPUT_INDEX
- **`level`**: Hierarchical level of the Ophint node, a number.
- **`children_inputs_mappings`**: Inputs/Outputs mapping for children hints.

Methods

`add`

```
add(
```

```
    arg,

    tag=None,

    name=None,

    aggregate=None,

    index_override=None

)
```

Return a wrapped tensor of an input tensor as an argument.

Args:

- **`arg`**: A TensorFlow tensor that should be considered an argument.
- **`tag`**: String tag to identify arguments that should be packed.
- **`name`**: Name of argument. This is included in the Identity hint op names.
- **`aggregate`**: Strategy to aggregate. Acceptable values are OpHint.AGGREGATE_FIRST, OpHint.AGGREGATE_LAST, and OpHint.AGGREGATE_STACK. Note, aggregate is only valid if tag is specified.
- **`index_override`**: Specify what input/output index should this be in the final stub. i.e. add(arg0, index=1); add(arg1, index=0) will make the final stub be as stub_func(inputs[arg1, arg0], outputs=[]) rather than the default call order based ordering.

Returns:

A tensor representing the wrapped argument.

Raises:

- **`ValueError`**: When indices are not consistent.

tf.compat.v1.lite.TFLiteConverter

Contents
- Class TFLiteConverter
- __init__
- Methods
  - convert

Class `TFLiteConverter`

Convert a TensorFlow model into `output_format`.

Defined in `lite/python/lite.py`.

This is used to convert from a TensorFlow GraphDef, SavedModel or tf.keras model into either a TFLite FlatBuffer or graph visualization.

Attributes:

- **`inference_type`**: Target data type of real-number arrays in the output file. Must be `{tf.float32, tf.uint8}`. If `optimzations` are provided, this parameter is ignored. (default tf.float32)
- **`inference_input_type`**: Target data type of real-number input arrays. Allows for a different type for input arrays. If an integer type is provided and `optimizations` are not used,`quantized_inputs_stats` must be provided. If `inference_type` is tf.uint8, signaling conversion to a fully quantized model from a quantization-aware trained input model,

then`inference_input_type` defaults to tf.uint8. In all other cases, `inference_input_type`defaults to tf.float32. Must be `{tf.float32, tf.uint8, tf.int8}`

- **`inference_output_type`**: Target data type of real-number output arrays. Allows for a different type for output arrays. If `inference_type` is tf.uint8, signaling conversion to a fully quantized model from a quantization-aware trained output model, then `inference_output_type` defaults to tf.uint8. In all other cases, `inference_output_type` must be tf.float32, an error will be thrown otherwise. Must be `{tf.float32, tf.uint8, tf.int8}`
- **`output_format`**: Output file format. Currently must be `{TFLITE, GRAPHVIZ_DOT}`. (default TFLITE)
- **`quantized_input_stats`**: Dict of strings representing input tensor names mapped to tuple of floats representing the mean and standard deviation of the training data (e.g., {"foo" : (0., 1.)}). Only need if `inference_input_type` is `QUANTIZED_UINT8`. real_input_value = (quantized_input_value - mean_value) / std_dev_value. (default {})
- **`default_ranges_stats`**: Tuple of integers representing (min, max) range values for all arrays without a specified range. Intended for experimenting with quantization via "dummy quantization". (default None)
- **`drop_control_dependency`**: Boolean indicating whether to drop control dependencies silently. This is due to TFLite not supporting control dependencies. (default True)
- **`reorder_across_fake_quant`**: Boolean indicating whether to reorder FakeQuant nodes in unexpected locations. Used when the location of the FakeQuant nodes is preventing graph transformations necessary to convert the graph. Results in a graph that differs from the quantized training graph, potentially causing differing arithmetic behavior. (default False)
- **`change_concat_input_ranges`**: Boolean to change behavior of min/max ranges for inputs and outputs of the concat operator for quantized models. Changes the ranges of concat operator overlap when true. (default False)
- **`allow_custom_ops`**: Boolean indicating whether to allow custom operations. When false any unknown operation is an error. When true, custom ops are created for any op that is unknown. The developer will need to provide these to the TensorFlow Lite runtime with a custom resolver. (default False)
- **`post_training_quantize`**: Deprecated. Please specify `[Optimize.DEFAULT]` for`optimizations` instead. Boolean indicating whether to quantize the weights of the converted float model. Model size will be reduced and there will be latency improvements (at the cost of accuracy). (default False)
- **`dump_graphviz_dir`**: Full filepath of folder to dump the graphs at various stages of processing GraphViz .dot files. Preferred over --output_format=GRAPHVIZ_DOT in order to keep the requirements of the output file. (default None)
- **`dump_graphviz_video`**: Boolean indicating whether to dump the graph after every graph transformation. (default False)
- **`target_ops`**: Deprecated. Please specify `target_spec.supported_ops` instead. Set of OpsSet options indicating which converter to use. (default set([OpsSet.TFLITE_BUILTINS]))
- **`target_spec`**: Experimental flag, subject to change. Specification of target device.
- **`optimizations`**: Experimental flag, subject to change. A list of optimizations to apply when converting the model. E.g. `[Optimize.DEFAULT]`
- **`representative_dataset`**: A representative dataset that can be used to generate input and output samples for the model. The converter can use the dataset to evaluate different optimizations.

Example usage:

```
# Converting a GraphDef from session.

converter = lite.TFLiteConverter.from_session(sess, in_tensors, out_tensors)

tflite_model = converter.convert()
```

```
open("converted_model.tflite", "wb").write(tflite_model)


# Converting a GraphDef from file.

converter = lite.TFLiteConverter.from_frozen_graph(

  graph_def_file, input_arrays, output_arrays)

tflite_model = converter.convert()

open("converted_model.tflite", "wb").write(tflite_model)


# Converting a SavedModel.

converter = lite.TFLiteConverter.from_saved_model(saved_model_dir)

tflite_model = converter.convert()

open("converted_model.tflite", "wb").write(tflite_model)


# Converting a tf.keras model.

converter = lite.TFLiteConverter.from_keras_model_file(keras_model)

tflite_model = converter.convert()

open("converted_model.tflite", "wb").write(tflite_model)
```

__init__

```
__init__(

    graph_def,

    input_tensors,

    output_tensors,

    input_arrays_with_shape=None,

    output_arrays=None
```

```
)
```

Constructor for TFLiteConverter.

Args:

- **`graph_def`**: Frozen TensorFlow GraphDef.
- **`input_tensors`**: List of input tensors. Type and shape are computed using `foo.shape` and `foo.dtype`.
- **`output_tensors`**: List of output tensors (only .name is used from this).
- **`input_arrays_with_shape`**: Tuple of strings representing input tensor names and list of integers representing input shapes (e.g., [("foo" : [1, 16, 16, 3])]). Use only when graph cannot be loaded into TensorFlow and when `input_tensors` and `output_tensors` are None. (default None)
- **`output_arrays`**: List of output tensors to freeze graph with. Use only when graph cannot be loaded into TensorFlow and when `input_tensors` and `output_tensors` are None. (default None)

Raises:

- **`ValueError`**: Invalid arguments.

Methods

`convert`

```
convert()
```

Converts a TensorFlow GraphDef based on instance variables.

Returns:

The converted data in serialized format. Either a TFLite Flatbuffer or a Graphviz graph depending on value in `output_format`.

Raises:

- **`ValueError`**: Input shape is not specified. None value for dimension in input_tensor.

`from_frozen_graph`

```
@classmethod

from_frozen_graph(

    cls,

    graph_def_file,

    input_arrays,

    output_arrays,

    input_shapes=None

)
```

Creates a TFLiteConverter class from a file containing a frozen GraphDef.

Args:

- `graph_def_file`: Full filepath of file containing frozen GraphDef.
- `input_arrays`: List of input tensors to freeze graph with.
- `output_arrays`: List of output tensors to freeze graph with.
- `input_shapes`: Dict of strings representing input tensor names to list of integers representing input shapes (e.g., {"foo" : [1, 16, 16, 3]}). Automatically determined when input shapes is None (e.g., {"foo" : None}). (default None)

Returns:
TFLiteConverter class.

Raises:

- `IOError`: File not found. Unable to parse input file.
- `ValueError`: The graph is not frozen. input_arrays or output_arrays contains an invalid tensor name. input_shapes is not correctly defined when required

from_keras_model_file

```
@classmethod

from_keras_model_file(

    cls,

    model_file,

    input_arrays=None,

    input_shapes=None,

    output_arrays=None,

    custom_objects=None

)
```

Creates a TFLiteConverter class from a tf.keras model file.

Args:

- `model_file`: Full filepath of HDF5 file containing the tf.keras model.
- `input_arrays`: List of input tensors to freeze graph with. Uses input arrays from SignatureDef when none are provided. (default None)
- `input_shapes`: Dict of strings representing input tensor names to list of integers representing input shapes (e.g., {"foo" : [1, 16, 16, 3]}). Automatically determined when input shapes is None (e.g., {"foo" : None}). (default None)
- `output_arrays`: List of output tensors to freeze graph with. Uses output arrays from SignatureDef when none are provided. (default None)
- `custom_objects`: Dict mapping names (strings) to custom classes or functions to be considered during model deserialization. (default None)

Returns:
TFLiteConverter class.

`from_saved_model`

```
@classmethod

from_saved_model(

    cls,

    saved_model_dir,

    input_arrays=None,

    input_shapes=None,

    output_arrays=None,

    tag_set=None,

    signature_key=None

)
```

Creates a TFLiteConverter class from a SavedModel.

Args:

- **`saved_model_dir`**: SavedModel directory to convert.
- **`input_arrays`**: List of input tensors to freeze graph with. Uses input arrays from SignatureDef when none are provided. (default None)
- **`input_shapes`**: Dict of strings representing input tensor names to list of integers representing input shapes (e.g., {"foo" : [1, 16, 16, 3]}). Automatically determined when input shapes is None (e.g., {"foo" : None}). (default None)
- **`output_arrays`**: List of output tensors to freeze graph with. Uses output arrays from SignatureDef when none are provided. (default None)
- **`tag_set`**: Set of tags identifying the MetaGraphDef within the SavedModel to analyze. All tags in the tag set must be present. (default set("serve"))
- **`signature_key`**: Key identifying SignatureDef containing inputs and outputs. (default DEFAULT_SERVING_SIGNATURE_DEF_KEY)

Returns:

TFLiteConverter class.

`from_session`

```
@classmethod

from_session(

    cls,

    sess,
```

```
    input_tensors,

    output_tensors

)
```

Creates a TFLiteConverter class from a TensorFlow Session.

Args:

- **`sess`**: TensorFlow Session.
- **`input_tensors`**: List of input tensors. Type and shape are computed using `foo.shape` and `foo.dtype`.
- **`output_tensors`**: List of output tensors (only .name is used from this).

Returns:
TFLiteConverter class.

`get_input_arrays`

```
get_input_arrays()
```

Returns a list of the names of the input tensors.

Returns:
List of strings.

tf.compat.v1.lite.TocoConverter

Class `TocoConverter`

Convert a TensorFlow model into `output_format` using TOCO.
Defined in `lite/python/lite.py`.
This class has been deprecated. Please use `lite.TFLiteConverter` instead.

Methods

`from_frozen_graph`

```
@classmethod

from_frozen_graph(

    cls,

    graph_def_file,
```

```
    input_arrays,

    output_arrays,

    input_shapes=None

)
```

Creates a TocoConverter class from a file containing a frozen graph. (deprecated)
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `lite.TFLiteConverter.from_frozen_graph` instead.

from_keras_model_file

```
@classmethod

from_keras_model_file(

    cls,

    model_file,

    input_arrays=None,

    input_shapes=None,

    output_arrays=None

)
```

Creates a TocoConverter class from a tf.keras model file. (deprecated)
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `lite.TFLiteConverter.from_keras_model_file` instead.

from_saved_model

```
@classmethod

from_saved_model(

    cls,

    saved_model_dir,

    input_arrays=None,

    input_shapes=None,
```

```
    output_arrays=None,

    tag_set=None,

    signature_key=None

)
```

Creates a TocoConverter class from a SavedModel. (deprecated)
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `lite.TFLiteConverter.from_saved_model` instead.

from_session

```
@classmethod

from_session(

    cls,

    sess,

    input_tensors,

    output_tensors

)
```

Creates a TocoConverter class from a TensorFlow Session. (deprecated)
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `lite.TFLiteConverter.from_session` instead.

tf.compat.v1.lite.toco_convert

Convert a model using TOCO. (deprecated)

```
tf.compat.v1.lite.toco_convert(

    input_data,

    input_tensors,

    output_tensors,

    *args,

    **kwargs

)
```

Defined in `lite/python/convert.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use **`lite.TFLiteConverter`** instead.

Typically this function is used to convert from TensorFlow GraphDef to TFLite. Conversion can be customized by providing arguments that are forwarded to `build_toco_convert_protos` (see documentation for details). This function has been deprecated. Please use `lite.TFLiteConverter`instead.

Args:

- **`input_data`**: Input data (i.e. often `sess.graph_def`),
- **`input_tensors`**: List of input tensors. Type and shape are computed using `foo.shape` and `foo.dtype`.
- **`output_tensors`**: List of output tensors (only .name is used from this).
- **`*args`**: See `build_toco_convert_protos`,
- **`**kwargs`**: See `build_toco_convert_protos`.

Returns:

The converted data. For example if TFLite was the destination, then this will be a tflite flatbuffer in a bytes array.

Raises:

Defined in `build_toco_convert_protos`.

Module: tf.compat.v1.lite.experimental

- **Contents**
- Modules
- Functions

Public API for tf.lite.experimental namespace.

Modules

`nn` module: Public API for tf.lite.experimental.nn namespace.

Functions

`convert_op_hints_to_stubs(...)`: Converts a graphdef with LiteOp hints into stub operations.
`get_potentially_supported_ops(...)`: Returns operations potentially supported by TensorFlow Lite.

tf.compat.v1.lite.experimental.convert_op_hints_to_stubs

Converts a graphdef with LiteOp hints into stub operations.

```
tf.compat.v1.lite.experimental.convert_op_hints_to_stubs(

    session=None,

    graph_def=None,

    write_callback=(lambda graph_def, comments: None)

)
```

Defined in `lite/python/op_hint.py`.
This is used to prepare for toco conversion of complex intrinsic usages. Note: only one of session or graph_def should be used, not both.

Args:

- **`session`**: A TensorFlow session that contains the graph to convert.
- **`graph_def`**: A graph def that we should convert.
- **`write_callback`**: A function pointer that can be used to write intermediate steps of graph transformation (optional).

Returns:
A new graphdef with all ops contained in OpHints being replaced by a single op call with the right parameters.

Raises:

- **`ValueError`**: If both session and graph_def are provided.

tf.compat.v1.lite.experimental.get_potentially_supported_ops

Returns operations potentially supported by TensorFlow Lite.

```
tf.compat.v1.lite.experimental.get_potentially_supported_ops()
```

Defined in `lite/experimental/tensorboard/ops_util.py`.
The potentially support list contains a list of ops that are partially or fully supported, which is derived by simply scanning op names to check whether they can be handled without real conversion and specific parameters.
Given that some ops may be partially supported, the optimal way to determine if a model's operations are supported is by converting using the TensorFlow Lite converter.

Returns:
A list of SupportedOp.

Module: tf.compat.v1.lite.experimental.nn

- **Contents**
- Classes
- Functions

Public API for tf.lite.experimental.nn namespace.

Classes

`class TFLiteLSTMCell`: Long short-term memory unit (LSTM) recurrent network cell.
`class TfLiteRNNCell`: The most basic RNN cell.

Functions

`dynamic_rnn(...)`: Creates a recurrent neural network specified by RNNCell `cell`.

tf.compat.v1.lite.experimental.nn.dynamic_rnn

Creates a recurrent neural network specified by RNNCell `cell`.

```
tf.compat.v1.lite.experimental.nn.dynamic_rnn(

    cell,
```

```
    inputs,

    sequence_length=None,

    initial_state=None,

    dtype=None,

    parallel_iterations=None,

    swap_memory=False,

    time_major=True,

    scope=None

)
```

Defined in `lite/experimental/examples/lstm/rnn.py`.
Performs fully dynamic unrolling of `inputs`.

Example:

```
# create a BasicRNNCell

rnn_cell = tf.compat.v1.nn.rnn_cell.BasicRNNCell(hidden_size)



# 'outputs' is a tensor of shape [batch_size, max_time, cell_state_size]



# defining initial state

initial_state = rnn_cell.zero_state(batch_size, dtype=tf.float32)



# 'state' is a tensor of shape [batch_size, cell_state_size]

outputs, state = tf.compat.v1.nn.dynamic_rnn(rnn_cell, input_data,

                                   initial_state=initial_state,

                                   dtype=tf.float32)
```

```
# create 2 LSTMCells

rnn_layers = [tf.compat.v1.nn.rnn_cell.LSTMCell(size) for size in [128, 256]]



# create a RNN cell composed sequentially of a number of RNNCells

multi_rnn_cell = tf.compat.v1.nn.rnn_cell.MultiRNNCell(rnn_layers)



# 'outputs' is a tensor of shape [batch_size, max_time, 256]

# 'state' is a N-tuple where N is the number of LSTMCells containing a

# tf.nn.rnn_cell.LSTMStateTuple for each cell

outputs, state = tf.compat.v1.nn.dynamic_rnn(cell=multi_rnn_cell,

                                   inputs=data,

                                   dtype=tf.float32)
```

Args:

- **`cell`**: An instance of RNNCell.
- **`inputs`**: The RNN inputs. If `time_major == False` (default), this must be a `Tensor` of shape: `[batch_size, max_time, ...]`, or a nested tuple of such elements. If `time_major == True`, this must be a `Tensor` of shape: `[max_time, batch_size, ...]`, or a nested tuple of such elements. This may also be a (possibly nested) tuple of Tensors satisfying this property. The first two dimensions must match across all the inputs, but otherwise the ranks and other shape components may differ. In this case, input to `cell` at each time-step will replicate the structure of these tuples, except for the time dimension (from which the time is taken). The input to `cell`at each time step will be a `Tensor` or (possibly nested) tuple of Tensors each with dimensions `[batch_size, ...]`.
- **`sequence_length`**: (optional) An int32/int64 vector sized `[batch_size]`. Used to copy-through state and zero-out outputs when past a batch element's sequence length. So it's more for performance than correctness.
- **`initial_state`**: (optional) An initial state for the RNN. If `cell.state_size` is an integer, this must be a `Tensor` of appropriate type and shape `[batch_size, cell.state_size]`. If `cell.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s] for s in cell.state_size`.
- **`dtype`**: (optional) The data type for the initial state and expected output. Required if initial_state is not provided or RNN state has a heterogeneous dtype.
- **`parallel_iterations`**: (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer.

- **`swap_memory`**: Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.
- **`time_major`**: The shape format of the `inputs` and `outputs` Tensors. If true, these `Tensors`must be shaped `[max_time, batch_size, depth]`. If false, these `Tensors` must be shaped `[batch_size, max_time, depth]`. Using `time_major = True` is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.
- **`scope`**: VariableScope for the created subgraph; defaults to "rnn".

Returns:

A pair (outputs, state) where:

- **`outputs`**: The RNN output `Tensor`.
  If time_major == False (default), this will be a `Tensor` shaped: `[batch_size, max_time, cell.output_size]`.
  If time_major == True, this will be a `Tensor` shaped: `[max_time, batch_size, cell.output_size]`.
  Note, if `cell.output_size` is a (possibly nested) tuple of integers or `TensorShape` objects, then `outputs` will be a tuple having the same structure as `cell.output_size`, containing Tensors having shapes corresponding to the shape data in `cell.output_size`.
- **`state`**: The final state. If `cell.state_size` is an int, this will be shaped `[batch_size, cell.state_size]`. If it is a `TensorShape`, this will be shaped `[batch_size] + cell.state_size`. If it is a (possibly nested) tuple of ints or `TensorShape`, this will be a tuple having the corresponding shapes. If cells are `LSTMCells` `state` will be a tuple containing a `LSTMStateTuple` for each cell.

Raises:

- **`TypeError`**: If `cell` is not an instance of RNNCell.
- **`ValueError`**: If inputs is None or an empty list.
- **`RuntimeError`**: If not using control flow v2.

tf.compat.v1.lite.experimental.nn.TFLiteLSTMCell

- Contents
- Class TFLiteLSTMCell
- __init__
- Properties
  - graph

Class `TFLiteLSTMCell`

Long short-term memory unit (LSTM) recurrent network cell.

Defined in `lite/experimental/examples/lstm/rnn_cell.py`.

This is used only for TfLite, it provides hints and it also makes the variables in the desired for the tflite ops (transposed and seaparated).

The default non-peephole implementation is based on:

https://pdfs.semanticscholar.org/1154/0131eae85b2e11d53df7f1360eeb6476e7f4.pdf

Felix Gers, Jurgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." IET, 850-855, 1999.

The peephole implementation is based on:

https://research.google.com/pubs/archive/43905.pdf

Hasim Sak, Andrew Senior, and Francoise Beaufays. "Long short-term memory recurrent neural network architectures for large scale acoustic modeling." INTERSPEECH, 2014.

The class uses optional peep-hole connections, optional cell clipping, and an optional projection layer.
Note that this cell is not optimized for performance. Please use `tf.contrib.cudnn_rnn.CudnnLSTM`for better performance on GPU, or `tf.contrib.rnn.LSTMBlockCell` and `tf.contrib.rnn.LSTMBlockFusedCell` for better performance on CPU.

`__init__`

```
__init__(

    num_units,

    use_peepholes=False,

    cell_clip=None,

    initializer=None,

    num_proj=None,

    proj_clip=None,

    num_unit_shards=None,

    num_proj_shards=None,

    forget_bias=1.0,

    state_is_tuple=True,

    activation=None,

    reuse=None,

    name=None,

    dtype=None

)
```

Initialize the parameters for an LSTM cell.

Args:

- **`num_units`**: int, The number of units in the LSTM cell.
- **`use_peepholes`**: bool, set True to enable diagonal/peephole connections.
- **`cell_clip`**: (optional) A float value, if provided the cell state is clipped by this value prior to the cell output activation.
- **`initializer`**: (optional) The initializer to use for the weight and projection matrices.

- **`num_proj`**: (optional) int, The output dimensionality for the projection matrices. If None, no projection is performed.
- **`proj_clip`**: (optional) A float value. If `num_proj > 0` and `proj_clip` is provided, then the projected values are clipped elementwise to within `[-proj_clip, proj_clip]`.
- **`num_unit_shards`**: Deprecated, will be removed by Jan. 2017. Use a variable_scope partitioner instead.
- **`num_proj_shards`**: Deprecated, will be removed by Jan. 2017. Use a variable_scope partitioner instead.
- **`forget_bias`**: Biases of the forget gate are initialized by default to 1 in order to reduce the scale of forgetting at the beginning of the training. Must set it manually to `0.0` when restoring from CudnnLSTM trained checkpoints.
- **`state_is_tuple`**: If True, accepted and returned states are 2-tuples of the `c_state` and `m_state`. If False, they are concatenated along the column axis. This latter behavior will soon be deprecated.
- **`activation`**: Activation function of the inner states. Default: `tanh`.
- **`reuse`**: (optional) Python boolean describing whether to reuse variables in an existing scope. If not `True`, and the existing scope already has the given variables, an error is raised.
- **`name`**: String, the name of the layer. Layers with the same name will share weights, but to avoid mistakes we require reuse=True in such cases.
- **`dtype`**: Default dtype of the layer (default of `None` means use the type of the first input). Required when `build` is called before `call`. When restoring from CudnnLSTM-trained checkpoints, use `CudnnCompatibleLSTMCell` instead.

Properties

`graph`

DEPRECATED FUNCTION

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Stop using this property because tf.layers layers no longer track their graph.

`output_size`

`scope_name`

`state_size`

Methods

`get_initial_state`

```
get_initial_state(

    inputs=None,

    batch_size=None,

    dtype=None

)
```

`zero_state`

```
zero_state(
```

```
    batch_size,

    dtype

)
```

Return zero-filled state tensor(s).

Args:

- **`batch_size`**: int, float, or unit Tensor representing the batch size.
- **`dtype`**: the data type to use for the state.

Returns:

If `state_size` is an int or TensorShape, then the return value is a `N-D` tensor of shape `[batch_size, state_size]` filled with zeros.
If `state_size` is a nested list or tuple, then the return value is a nested list or tuple (of the same structure) of `2-D` tensors with the shapes `[batch_size, s]` for each s in `state_size`.

tf.compat.v1.lite.experimental.nn.TfLiteRNNCell

- Contents
- Class TfLiteRNNCell
- __init__
- Properties
  - graph

Class `TfLiteRNNCell`

The most basic RNN cell.
Defined in `lite/experimental/examples/lstm/rnn_cell.py`.
This is used only for TfLite, it provides hints and it also makes the variables in the desired for the tflite ops.

`__init__`

```
__init__(

    num_units,

    activation=None,

    reuse=None,

    name=None,

    dtype=None,

    **kwargs

)
```

Initializes the parameters for an RNN cell.

Args:

- `num_units`: int, The number of units in the RNN cell.
- `activation`: Nonlinearity to use. Default: `tanh`. It could also be string that is within Keras activation function names.
- `reuse`: (optional) Python boolean describing whether to reuse variables in an existing scope. Raises an error if not `True` and the existing scope already has the given variables.
- `name`: String, the name of the layer. Layers with the same name will share weights, but to avoid mistakes we require reuse=True in such cases.
- `dtype`: Default dtype of the layer (default of `None` means use the type of the first input). Required when `build` is called before `call`.
- `**kwargs`: Dict, keyword named properties for common layer attributes, like `trainable` etc when constructing the cell from configs of get_config().

Raises:

- `ValueError`: If the existing scope already has the given variables.

Properties

`graph`

DEPRECATED FUNCTION

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Stop using this property because tf.layers layers no longer track their graph.

`output_size`

`scope_name`

`state_size`

Methods

`get_initial_state`

```
get_initial_state(

    inputs=None,

    batch_size=None,

    dtype=None

)
```

`zero_state`

```
zero_state(

    batch_size,

    dtype

)
```

Return zero-filled state tensor(s).

Args:

- `batch_size`: int, float, or unit Tensor representing the batch size.
- `dtype`: the data type to use for the state.

Returns:

If `state_size` is an int or TensorShape, then the return value is a `N-D` tensor of shape `[batch_size, state_size]` filled with zeros.
If `state_size` is a nested list or tuple, then the return value is a nested list or tuple (of the same structure) of `2-D` tensors with the shapes `[batch_size, s]` for each s in `state_size`.

Module: tf.lite

- Contents
- Classes

Public API for tf.lite namespace.

Classes

`class Interpreter`: Interpreter interface for TensorFlow Lite Models.
`class OpsSet`: Enum class defining the sets of ops available to generate TFLite models.
`class Optimize`: Enum defining the optimizations to apply when generating tflite graphs.
`class RepresentativeDataset`: Representative dataset to evaluate optimizations.
`class TFLiteConverter`: Converts a TensorFlow model into TensorFlow Lite model.
`class TargetSpec`: Specification of target device.

tf.lite.Interpreter

- Contents
- Class Interpreter
  - Aliases:
- __init__
- Methods

Class `Interpreter`

Interpreter interface for TensorFlow Lite Models.

Aliases:

- Class `tf.compat.v1.lite.Interpreter`
- Class `tf.compat.v2.lite.Interpreter`
- Class `tf.lite.Interpreter`

Defined in `lite/python/interpreter.py`.
This makes the TensorFlow Lite interpreter accessible in Python. It is possible to use this interpreter in a multithreaded Python environment, but you must be sure to call functions of a particular instance from only one thread at a time. So if you want to have 4 threads running different inferences simultaneously, create an interpreter for each one as thread-local data. Similarly, if you are calling invoke() in one thread on a single interpreter but you want to use tensor() on another thread once it is done, you must use a synchronization primitive between the threads to ensure invoke has returned before calling tensor().

`__init__`

```
__init__(
```

```
    model_path=None,

    model_content=None

)
```

Constructor.

Args:

- **`model_path`**: Path to TF-Lite Flatbuffer file.
- **`model_content`**: Content of model.

Raises:

- **`ValueError`**: If the interpreter was unable to create.

Methods

`allocate_tensors`

```
allocate_tensors()
```

`get_input_details`

```
get_input_details()
```

Gets model input details.

Returns:

A list of input details.

`get_output_details`

```
get_output_details()
```

Gets model output details.

Returns:

A list of output details.

`get_tensor`

```
get_tensor(tensor_index)
```

Gets the value of the input tensor (get a copy).

If you wish to avoid the copy, use `tensor()`. This function cannot be used to read intermediate results.

Args:

- **`tensor_index`**: Tensor index of tensor to get. This value can be gotten from the 'index' field in get_output_details.

Returns:

a numpy array.

`get_tensor_details`

```
get_tensor_details()
```

Gets tensor details for every tensor with valid tensor details.
Tensors where required information about the tensor is not found are not added to the list. This includes temporary tensors without a name.

Returns:
A list of dictionaries containing tensor information.

`invoke`

```
invoke()
```

Invoke the interpreter.
Be sure to set the input sizes, allocate tensors and fill values before calling this. Also, note that this function releases the GIL so heavy computation can be done in the background while the Python interpreter continues. No other function on this object should be called while the invoke() call has not finished.

Raises:

- **`ValueError`**: When the underlying interpreter fails raise ValueError.

`reset_all_variables`

```
reset_all_variables()
```

`resize_tensor_input`

```
resize_tensor_input(

    input_index,

    tensor_size

)
```

Resizes an input tensor.

Args:

- **`input_index`**: Tensor index of input to set. This value can be gotten from the 'index' field in get_input_details.
- **`tensor_size`**: The tensor_shape to resize the input to.

Raises:

- **`ValueError`**: If the interpreter could not resize the input tensor.

`set_tensor`

```
set_tensor(
```

```
    tensor_index,

    value

)
```

Sets the value of the input tensor. Note this copies data in `value`.
If you want to avoid copying, you can use the `tensor()` function to get a numpy buffer pointing to the input buffer in the tflite interpreter.

Args:

- **`tensor_index`**: Tensor index of tensor to set. This value can be gotten from the 'index' field in get_input_details.
- **`value`**: Value of tensor to set.

Raises:

- **`ValueError`**: If the interpreter could not set the tensor.

tensor

```
tensor(tensor_index)
```

Returns function that gives a numpy view of the current tensor buffer.
This allows reading and writing to this tensors w/o copies. This more closely mirrors the C++ Interpreter class interface's tensor() member, hence the name. Be careful to not hold these output references through calls to `allocate_tensors()` and `invoke()`. This function cannot be used to read intermediate results.

Usage:

```
interpreter.allocate_tensors()

input = interpreter.tensor(interpreter.get_input_details()[0]["index"])

output = interpreter.tensor(interpreter.get_output_details()[0]["index"])

for i in range(10):

  input().fill(3.)

  interpreter.invoke()

  print("inference %s" % output())
```

Notice how this function avoids making a numpy array directly. This is because it is important to not hold actual numpy views to the data longer than necessary. If you do, then the interpreter can no longer be invoked, because it is possible the interpreter would resize and invalidate the referenced tensors. The NumPy API doesn't allow any mutability of the the underlying buffers.

WRONG:

```
input = interpreter.tensor(interpreter.get_input_details()[0]["index"])()
```

```
output = interpreter.tensor(interpreter.get_output_details()[0]["index"])()

interpreter.allocate_tensors()  # This will throw RuntimeError

for i in range(10):

  input.fill(3.)

  interpreter.invoke()  # this will throw RuntimeError since input,output

```

Args:

- **`tensor_index`**: Tensor index of tensor to get. This value can be gotten from the 'index' field in get_output_details.

Returns:
A function that can return a new numpy array pointing to the internal TFLite tensor state at any point. It is safe to hold the function forever, but it is not safe to hold the numpy array forever.

tf.lite.OpsSet

Class `OpsSet`

Enum class defining the sets of ops available to generate TFLite models.

Aliases:

- Class `tf.compat.v1.lite.OpsSet`
- Class `tf.compat.v2.lite.OpsSet`
- Class `tf.lite.OpsSet`

Defined in `lite/python/convert.py`.
WARNING: Experimental interface, subject to change.

Class Members

- `SELECT_TF_OPS`
- `TFLITE_BUILTINS`
- `TFLITE_BUILTINS_INT8`

tf.lite.Optimize

Class `Optimize`

Enum defining the optimizations to apply when generating tflite graphs.

Aliases:

- Class `tf.compat.v1.lite.Optimize`
- Class `tf.compat.v2.lite.Optimize`

- Class `tf.lite.Optimize`
  Defined in `lite/python/lite.py`.
  Some optimizations may come at the cost of accuracy.

  ## Class Members

- `DEFAULT`
- `OPTIMIZE_FOR_LATENCY`
- `OPTIMIZE_FOR_SIZE`

tf.lite.RepresentativeDataset

- Contents
- Class RepresentativeDataset
  - Aliases:
- __init__

Class `RepresentativeDataset`

Representative dataset to evaluate optimizations.

Aliases:

- Class `tf.compat.v1.lite.RepresentativeDataset`
- Class `tf.compat.v2.lite.RepresentativeDataset`
- Class `tf.lite.RepresentativeDataset`
  Defined in `lite/python/lite.py`.
  A representative dataset that can be used to evaluate optimizations by the converter. E.g. converter can use these examples to estimate (min, max) ranges by calibrating the model on inputs. This can allow converter to quantize a converted floating point model.

`__init__`

```
__init__(input_gen)
```

Creates a representative dataset.

Args:

- **`input_gen`**: an input generator that can be used to generate input samples for the model. This must be a callable object that returns an object that supports the `iter()` protocol (e.g. a generator function). The elements generated must have same type and shape as inputs to the model.

tf.lite.TargetSpec

- Contents
- Class TargetSpec
  - Aliases:
- __init__

Class `TargetSpec`

Specification of target device.

Aliases:

- Class `tf.compat.v1.lite.TargetSpec`
- Class `tf.compat.v2.lite.TargetSpec`
- Class `tf.lite.TargetSpec`
  Defined in `lite/python/lite.py`.
  Details about target device. Converter optimizes the generated model for specific device.

Attributes:

- `supported_ops`: Experimental flag, subject to change. Set of OpsSet options supported by the device. (default set([OpsSet.TFLITE_BUILTINS]))

`__init__`

```
__init__(supported_ops=None)
```

Module: tf.compat.v1.train / tf.train

- **Contents**
- Modules
- Classes
- Functions

Support for training models.
See the Training guide.

Modules

`experimental` module: Public API for tf.train.experimental namespace.
`queue_runner` module: Public API for tf.train.queue_runner namespace.

Classes

`class AdadeltaOptimizer`: Optimizer that implements the Adadelta algorithm.
`class AdagradDAOptimizer`: Adagrad Dual Averaging algorithm for sparse linear models.
`class AdagradOptimizer`: Optimizer that implements the Adagrad algorithm.
`class AdamOptimizer`: Optimizer that implements the Adam algorithm.
`class BytesList`
`class Checkpoint`: Groups trackable objects, saving and restoring them.
`class CheckpointManager`: Deletes old checkpoints.
`class CheckpointSaverHook`: Saves checkpoints every N steps or seconds.
`class CheckpointSaverListener`: Interface for listeners that take action before or after checkpoint save.
`class ChiefSessionCreator`: Creates a tf.compat.v1.Session for a chief.
`class ClusterDef`
`class ClusterSpec`: Represents a cluster as a set of "tasks", organized into "jobs".
`class Coordinator`: A coordinator for threads.
`class Example`
`class ExponentialMovingAverage`: Maintains moving averages of variables by employing an exponential decay.
`class Feature`
`class FeatureList`
`class FeatureLists`
`class Features`
`class FeedFnHook`: Runs `feed_fn` and sets the `feed_dict` accordingly.
`class FinalOpsHook`: A hook which evaluates `Tensors` at the end of a session.
`class FloatList`
`class FtrlOptimizer`: Optimizer that implements the FTRL algorithm.
`class GlobalStepWaiterHook`: Delays execution until global step reaches `wait_until_step`.
`class GradientDescentOptimizer`: Optimizer that implements the gradient descent algorithm.
`class Int64List`
`class JobDef`
`class LoggingTensorHook`: Prints the given tensors every N local steps, every N seconds, or at end.
`class LooperThread`: A thread that runs code repeatedly, optionally on a timer.
`class MomentumOptimizer`: Optimizer that implements the Momentum algorithm.

class MonitoredSession: Session-like object that handles initialization, recovery and hooks.
class NanLossDuringTrainingError
class NanTensorHook: Monitors the loss tensor and stops training if loss is NaN.
class Optimizer: Base class for optimizers.
class ProfilerHook: Captures CPU/GPU profiling information every N steps or seconds.
class ProximalAdagradOptimizer: Optimizer that implements the Proximal Adagrad algorithm.
class ProximalGradientDescentOptimizer: Optimizer that implements the proximal gradient descent algorithm.
class QueueRunner: Holds a list of enqueue operations for a queue, each to be run in a thread.
class RMSPropOptimizer: Optimizer that implements the RMSProp algorithm.
class Saver: Saves and restores variables.
class SaverDef
class Scaffold: Structure to create or gather pieces commonly needed to train a model.
class SecondOrStepTimer: Timer that triggers at most once every N seconds or once every N steps.
class SequenceExample
class Server: An in-process TensorFlow server, for use in distributed training.
class ServerDef
class SessionCreator: A factory for tf.Session.
class SessionManager: Training helper that restores from checkpoint and creates session.
class SessionRunArgs: Represents arguments to be added to a `Session.run()` call.
class SessionRunContext: Provides information about the `session.run()` call being made.
class SessionRunHook: Hook to extend calls to MonitoredSession.run().
class SessionRunValues: Contains the results of `Session.run()`.
class SingularMonitoredSession: Session-like object that handles initialization, restoring, and hooks.
class StepCounterHook: Hook that counts steps per second.
class StopAtStepHook: Hook that requests stop at a specified step.
class SummarySaverHook: Saves summaries every N steps.
class Supervisor: A training helper that checkpoints models and computes summaries.
class SyncReplicasOptimizer: Class to synchronize, aggregate gradients and pass them to the optimizer.
class VocabInfo: Vocabulary information for warm-starting.
class WorkerSessionCreator: Creates a tf.compat.v1.Session for a worker.

Functions

MonitoredTrainingSession(...): Creates a `MonitoredSession` for training.
NewCheckpointReader(...)
add_queue_runner(...): Adds a `QueueRunner` to a collection in the graph. (deprecated)
assert_global_step(...): Asserts `global_step_tensor` is a scalar int `Variable` or `Tensor`.
basic_train_loop(...): Basic loop to train a model.
batch(...): Creates batches of tensors in `tensors`. (deprecated)
batch_join(...): Runs a list of tensors to fill a queue to create batches of examples. (deprecated)
checkpoint_exists(...): Checks whether a V1 or V2 checkpoint exists with the specified prefix. (deprecated)
checkpoints_iterator(...): Continuously yield new checkpoint files as they appear.
cosine_decay(...): Applies cosine decay to the learning rate.
cosine_decay_restarts(...): Applies cosine decay with restarts to the learning rate.
create_global_step(...): Create global step tensor in graph.
do_quantize_training_on_graphdef(...): A general quantization scheme is being developed in `tf.contrib.quantize`. (deprecated)
exponential_decay(...): Applies exponential decay to the learning rate.

export_meta_graph(...): Returns MetaGraphDef proto.
generate_checkpoint_state_proto(...): Generates a checkpoint state proto.
get_checkpoint_mtimes(...): Returns the mtimes (modification timestamps) of the checkpoints. (deprecated)
get_checkpoint_state(...): Returns CheckpointState proto from the "checkpoint" file.
get_global_step(...): Get the global step tensor.
get_or_create_global_step(...): Returns and create (if necessary) the global step tensor.
global_step(...): Small helper to get the global step.
import_meta_graph(...): Recreates a Graph saved in a MetaGraphDef proto.
init_from_checkpoint(...): Replaces tf.Variable initializers so they load from a checkpoint file.
input_producer(...): Output the rows of input_tensor to a queue for an input pipeline. (deprecated)
inverse_time_decay(...): Applies inverse time decay to the initial learning rate.
latest_checkpoint(...): Finds the filename of latest saved checkpoint file.
limit_epochs(...): Returns tensor num_epochs times and then raises an OutOfRange error. (deprecated)
linear_cosine_decay(...): Applies linear cosine decay to the learning rate.
list_variables(...): Returns list of all variables in the checkpoint.
load_checkpoint(...): Returns CheckpointReader for checkpoint found in ckpt_dir_or_file.
load_variable(...): Returns the tensor value of the given variable in the checkpoint.
match_filenames_once(...): Save the list of files matching pattern, so it is only computed once.
maybe_batch(...): Conditionally creates batches of tensors based on keep_input. (deprecated)
maybe_batch_join(...): Runs a list of tensors to conditionally fill a queue to create batches. (deprecated)
maybe_shuffle_batch(...): Creates batches by randomly shuffling conditionally-enqueued tensors. (deprecated)
maybe_shuffle_batch_join(...): Create batches by randomly shuffling conditionally-enqueued tensors. (deprecated)
natural_exp_decay(...): Applies natural exponential decay to the initial learning rate.
noisy_linear_cosine_decay(...): Applies noisy linear cosine decay to the learning rate.
piecewise_constant(...): Piecewise constant from boundaries and interval values.
piecewise_constant_decay(...): Piecewise constant from boundaries and interval values.
polynomial_decay(...): Applies a polynomial decay to the learning rate.
range_input_producer(...): Produces the integers from 0 to limit-1 in a queue. (deprecated)
remove_checkpoint(...): Removes a checkpoint given by checkpoint_prefix. (deprecated)
replica_device_setter(...): Return a device function to use when building a Graph for replicas.
sdca_fprint(...): Computes fingerprints of the input strings.
sdca_optimizer(...): Distributed version of Stochastic Dual Coordinate Ascent (SDCA) optimizer for
sdca_shrink_l1(...): Applies L1 regularization shrink step on the parameters.
shuffle_batch(...): Creates batches by randomly shuffling tensors. (deprecated)
shuffle_batch_join(...): Create batches by randomly shuffling tensors. (deprecated)
slice_input_producer(...): Produces a slice of each Tensor in tensor_list. (deprecated)
start_queue_runners(...): Starts all queue runners collected in the graph. (deprecated)
string_input_producer(...): Output strings (e.g. filenames) to a queue for an input pipeline. (deprecated)
summary_iterator(...): An iterator for reading Event protocol buffers from an event file.
update_checkpoint_state(...): Updates the content of the 'checkpoint' file. (deprecated)
warm_start(...): Warm-starts a model using the given settings.

`write_graph(...)`: Writes a graph proto to a file.

tf.compat.v1.train.AdadeltaOptimizer

- **Contents**
- Class AdadeltaOptimizer
- __init__
- Methods
  - apply_gradients

Class `AdadeltaOptimizer`

Optimizer that implements the Adadelta algorithm.
Inherits From: `Optimizer`
Defined in `python/training/adadelta.py`.
See M. D. Zeiler (pdf)

`__init__`

```
__init__(
    learning_rate=0.001,
    rho=0.95,
    epsilon=1e-08,
    use_locking=False,
    name='Adadelta'
)
```

Construct a new Adadelta optimizer.

Args:

- **`learning_rate`**: A `Tensor` or a floating point value. The learning rate. To match the exact form in the original paper use 1.0.
- **`rho`**: A `Tensor` or a floating point value. The decay rate.
- **`epsilon`**: A `Tensor` or a floating point value. A constant epsilon used to better conditioning the grad update.
- **`use_locking`**: If `True` use locks for update operations.
- **`name`**: Optional name prefix for the operations created when applying gradients. Defaults to "Adadelta".

Eager Compatibility
When eager execution is enabled, `learning_rate`, `rho`, and `epsilon` can each be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

compute_gradients

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility

When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

get_slot

```
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.

get_slot_names

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

minimize

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.AdagradDAOptimizer

- **Contents**
- Class AdagradDAOptimizer
- __init__
- Methods
  - apply_gradients

Class `AdagradDAOptimizer`

Adagrad Dual Averaging algorithm for sparse linear models.
Inherits From: `Optimizer`
Defined in `python/training/adagrad_da.py`.
See this paper.

This optimizer takes care of regularization of unseen features in a mini batch by updating them when they are seen with a closed form update rule that is equivalent to having updated them on every mini-batch.
AdagradDA is typically used when there is a need for large sparsity in the trained model. This optimizer only guarantees sparsity for linear models. Be careful when using AdagradDA for deep networks as it will require careful initialization of the gradient accumulators for it to train.

`__init__`

```
__init__(
    learning_rate,
    global_step,
    initial_gradient_squared_accumulator_value=0.1,
    l1_regularization_strength=0.0,
    l2_regularization_strength=0.0,
    use_locking=False,
    name='AdagradDA'
)
```

Construct a new AdagradDA optimizer.

Args:

- **`learning_rate`**: A `Tensor` or a floating point value. The learning rate.
- **`global_step`**: A `Tensor` containing the current training step number.
- **`initial_gradient_squared_accumulator_value`**: A floating point value. Starting value for the accumulators, must be positive.
- **`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 "AdagradDA".

Raises:

- **`ValueError`**: If the `initial_gradient_squared_accumulator_value` is invalid.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

compute_gradients

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

variables

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.AdagradOptimizer

- **Contents**
- Class AdagradOptimizer
- __init__
- Methods
  - apply_gradients

Class `AdagradOptimizer`

Optimizer that implements the Adagrad algorithm.
Inherits From: `Optimizer`
Defined in `python/training/adagrad.py`.
See this paper or this intro.

__init__

```
__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.

Eager Compatibility

When eager execution is enabled, `learning_rate` can be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
```

```
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility

When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

get_slot

```
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.

get_slot_names

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

minimize

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

variables

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.AdamOptimizer

- **Contents**

Class `AdamOptimizer`

Optimizer that implements the Adam algorithm.
Inherits From: [`Optimizer`]()
Defined in [`python/training/adam.py`]().

Used in the guide:

- [Convert Your Existing Code to TensorFlow 2.0]()

See [Kingma et al., 2014]() ([pdf]()).

__init__

```
__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:

$$m0:=0 \text{(Initialize initial 1st moment vector)}$$
$$v0:=0 \text{(Initialize initial 2nd moment vector)}$$
$$t:=0 \text{(Initialize timestep)}$$

The update rule for `variable` with gradient `g` uses an optimization described at the end of section 2 of the paper:

$$t:=t+1$$
$$lrt:=learning\_rate * 1-beta2t/(1-beta1t)$$
$$mt:=beta1 * mt-1+(1-beta1) * g$$

$$vt:=beta2*vt-1+(1-beta2)*g*g$$
$$variable:=variable-lrt*mt/(vt+\epsilon)$$

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. Note that since AdamOptimizer uses the formulation just before Section 2.1 of the Kingma and Ba paper rather than the formulation in Algorithm 1, the "epsilon" referred to here is "epsilon hat" in the paper.
The sparse implementation of this algorithm (used when the gradient is an IndexedSlices object, typically because of `tf.gather` or an embedding lookup in the forward pass) does apply momentum to variable slices even if they were not used in the forward pass (meaning they have a gradient equal to zero). Momentum decay (beta1) is also applied to the entire momentum accumulator. This means that the sparse behavior is equivalent to the dense behavior (in contrast to some momentum implementations which ignore momentum unless a variable slice was actually used).

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. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper.
- **`use_locking`**: If True use locks for update operations.
- **`name`**: Optional name for the operations created when applying gradients. Defaults to "Adam".

@compatibility(eager) When eager execution is enabled, `learning_rate`, `beta1`, `beta2`, and `epsilon` can each be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.
@end_compatibility

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

`get_name`

```
get_name()
```

`get_slot`

```
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.

get_slot_names

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:

A list of strings.

minimize

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.add_queue_runner

- **Contents**
- Aliases:

Adds a `QueueRunner` to a collection in the graph. (deprecated)

Aliases:

- `tf.compat.v1.train.add_queue_runner`
- `tf.compat.v1.train.queue_runner.add_queue_runner`

```
tf.compat.v1.train.add_queue_runner(
    qr,
    collection=tf.GraphKeys.QUEUE_RUNNERS
)
```

Defined in `python/training/queue_runner_impl.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: To construct input pipelines, use the **`tf.data`** module.
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.compat.v1.train.assert_global_step

Asserts `global_step_tensor` is a scalar int `Variable` or `Tensor`.

```
tf.compat.v1.train.assert_global_step(global_step_tensor)
```

Defined in `python/training/training_util.py`.

Args:

- **`global_step_tensor`**: `Tensor` to test.

tf.compat.v1.train.basic_train_loop

Basic loop to train a model.

```
tf.compat.v1.train.basic_train_loop(
    supervisor,
    train_step_fn,
    args=None,
    kwargs=None,
    master=''
)
```

Defined in `python/training/basic_loops.py`.

Calls `train_step_fn` in a loop to train a model. The function is called as:

```
train_step_fn(session, *args, **kwargs)
```

It is passed a `tf.compat.v1.Session` in addition to `args` and `kwargs`. The function typically runs one training step in the session.

Args:

- **`supervisor`**: `tf.compat.v1.train.Supervisor` to run the training services.
- **`train_step_fn`**: Callable to execute one training step. Called repeatedly as`train_step_fn(session, *args **kwargs)`.
- **`args`**: Optional positional arguments passed to `train_step_fn`.
- **`kwargs`**: Optional keyword arguments passed to `train_step_fn`.
- **`master`**: Master to use to create the training session. Defaults to `""` which causes the session to be created in the local process.

tf.compat.v1.train.batch

Creates batches of tensors in `tensors`. (deprecated)

```
tf.compat.v1.train.batch(
    tensors,
    batch_size,
    num_threads=1,
    capacity=32,
    enqueue_many=False,
    shapes=None,
    dynamic_pad=False,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`.
Use `tf.data.Dataset.batch(batch_size)` (or `padded_batch(...)` if `dynamic_pad=True`).
The argument `tensors` can be a list or a dictionary of tensors. The value returned by the function will be of the same type as `tensors`.
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`, `tensors` 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`, `tensors` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors` 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.: If `dynamic_pad` is `False`, you must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensors` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.
If `dynamic_pad` is `True`, it is sufficient that the *rank* of the tensors is known, but individual dimensions may have shape `None`. In this case, for each enqueue the dimensions with value `None`may have a variable length; upon dequeue, the output tensors will be padded on the right to the maximum shape of the tensors in the current minibatch. For numbers, this padding takes value 0. For strings, this padding is the empty string. See `PaddingFIFOQueue` for more info.
If `allow_smaller_final_batch` is `True`, a smaller batch value than `batch_size` is returned when the queue is closed and there are not enough elements to fill the batch, otherwise the pending elements are discarded. In addition, all output tensors' static shapes, as accessed via the `shape`property will have a first `Dimension` value of `None`, and operations that depend on fixed batch_size would fail.

Args:

- **`tensors`**: The list or dictionary of tensors to enqueue.
- **`batch_size`**: The new batch size pulled from the queue.
- **`num_threads`**: The number of threads enqueuing `tensors`. The batching will be nondeterministic if `num_threads > 1`.
- **`capacity`**: An integer. The maximum number of elements in the queue.
- **`enqueue_many`**: Whether each tensor in `tensors` is a single example.
- **`shapes`**: (Optional) The shapes for each example. Defaults to the inferred shapes for `tensors`.
- **`dynamic_pad`**: Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (Optional). If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same types as `tensors` (except if the input is a list of one element, then it returns a tensor, not a list).

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

Eager Compatibility

Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.batch_join

Runs a list of tensors to fill a queue to create batches of examples. (deprecated)

```
tf.compat.v1.train.batch_join(
    tensors_list,
    batch_size,
    capacity=32,
    enqueue_many=False,
    shapes=None,
    dynamic_pad=False,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.interleave(...).batch(batch_size)` (or `padded_batch(...)` if `dynamic_pad=True`).

The `tensors_list` argument is a list of tuples of tensors, or a list of dictionaries of tensors. Each element in the list is treated similarly to the `tensors` argument of `tf.compat.v1.train.batch()`.

WARNING: This function is nondeterministic, since it starts a separate thread for each tensor.

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(tensors_list)` threads will be started, with thread `i` enqueuing the tensors from`tensors_list[i]`. `tensors_list[i1][j]` must match `tensors_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensors_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`, `tensors_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors_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.: If `dynamic_pad` is `False`, you must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensors_list` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

If `dynamic_pad` is `True`, it is sufficient that the *rank* of the tensors is known, but individual dimensions may have value `None`. In this case, for each enqueue the dimensions with

value `None`may have a variable length; upon dequeue, the output tensors will be padded on the right to the maximum shape of the tensors in the current minibatch. For numbers, this padding takes value 0. For strings, this padding is the empty string. See `PaddingFIFOQueue` for more info.
If `allow_smaller_final_batch` is `True`, a smaller batch value than `batch_size` is returned when the queue is closed and there are not enough elements to fill the batch, otherwise the pending elements are discarded. In addition, all output tensors' static shapes, as accessed via the `shape`property will have a first `Dimension` value of `None`, and operations that depend on fixed batch_size would fail.

Args:

- **`tensors_list`**: A list of tuples or dictionaries 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]`.
- **`dynamic_pad`**: Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (Optional) If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:
A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensor_list_list`.

Eager Compatibility
Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.Checkpoint

- **Contents**
- Class Checkpoint
- __init__
- Properties
  - save_counter

Class `Checkpoint`

Groups trackable objects, saving and restoring them.
Defined in `python/training/tracking/util.py`.
`Checkpoint`'s constructor accepts keyword arguments whose values are types that contain trackable state, such as `tf.compat.v1.train.Optimizer` implementations, `tf.Variable`, `tf.keras.Layer`implementations, or `tf.keras.Model` implementations. It saves these values with a checkpoint, and maintains a `save_counter` for numbering checkpoints.
Example usage when graph building:

```
import tensorflow as tf
import os
```

```
checkpoint_directory = "/tmp/training_checkpoints"
checkpoint_prefix = os.path.join(checkpoint_directory, "ckpt")

checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
status = checkpoint.restore(tf.train.latest_checkpoint(checkpoint_directory))
train_op = optimizer.minimize( ... )
status.assert_consumed()  # Optional sanity checks.
with tf.compat.v1.Session() as session:
  # Use the Session to restore variables, or initialize them if
  # tf.train.latest_checkpoint returned None.
  status.initialize_or_restore(session)
  for _ in range(num_training_steps):
    session.run(train_op)
  checkpoint.save(file_prefix=checkpoint_prefix)
```

Example usage with eager execution enabled:

```
import tensorflow as tf
import os

tf.compat.v1.enable_eager_execution()

checkpoint_directory = "/tmp/training_checkpoints"
checkpoint_prefix = os.path.join(checkpoint_directory, "ckpt")

checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
status = checkpoint.restore(tf.train.latest_checkpoint(checkpoint_directory))
for _ in range(num_training_steps):
  optimizer.minimize( ... )  # Variables will be restored on creation.
status.assert_consumed()  # Optional sanity checks.
checkpoint.save(file_prefix=checkpoint_prefix)
```

`Checkpoint.save` and `Checkpoint.restore` write and read object-based checkpoints, in contrast to `tf.compat.v1.train.Saver` which writes and reads `variable.name` based checkpoints. Object-based checkpointing saves a graph of dependencies between Python objects (`Layer`s, `Optimizer`s, `Variable`s, etc.) with named edges, and this graph is used to match variables when restoring a checkpoint. It can be more robust to changes in the Python program, and helps to support restore-on-create for variables when executing eagerly.

Prefer `tf.train.Checkpoint` over `tf.compat.v1.train.Saver` for new code.

`Checkpoint` objects have dependencies on the objects passed as keyword arguments to their constructors, and each dependency is given a name that is identical to the name of the keyword argument for which it was created. TensorFlow classes like `Layer`s and `Optimizer`s will automatically add dependencies on their variables (e.g. "kernel" and "bias" for`tf.keras.layers.Dense`). Inheriting from `tf.keras.Model` makes managing dependencies easy in user-defined classes, since `Model` hooks into attribute assignment. For example:

```
class Regress(tf.keras.Model):

  def __init__(self):
    super(Regress, self).__init__()
    self.input_transform = tf.keras.layers.Dense(10)
```

```
    # ...

  def call(self, inputs):
    x = self.input_transform(inputs)
    # ...
```

This `Model` has a dependency named "input_transform" on its `Dense` layer, which in turn depends on its variables. As a result, saving an instance of `Regress` using `tf.train.Checkpoint` will also save all the variables created by the `Dense` layer.
When variables are assigned to multiple workers, each worker writes its own section of the checkpoint. These sections are then merged/re-indexed to behave as a single checkpoint. This avoids copying all variables to one worker, but does require that all workers see a common filesystem.
While `tf.keras.Model.save_weights` and `tf.train.Checkpoint.save` save in the same format, note that the root of the resulting checkpoint is the object the save method is attached to. This means saving a `tf.keras.Model` using `save_weights` and loading into a `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not match the `Model`'s variables. See the guide to training checkpoints for details.
Prefer `tf.train.Checkpoint` over `tf.keras.Model.save_weights` for training checkpoints.

Attributes:

- **`save_counter`**: Incremented when `save()` is called. Used to number checkpoints.

`__init__`

```
__init__(**kwargs)
```

Group objects into a training checkpoint.

Args:

- **`**kwargs`**: Keyword arguments are set as attributes of this object, and are saved with the checkpoint. Values must be trackable objects.

Raises:

- **`ValueError`**: If objects in `kwargs` are not trackable.

Properties

`save_counter`

An integer variable which starts at zero and is incremented on save.
Used to number checkpoints.

Returns:

The save counter variable.

Methods

`restore`

```
restore(save_path)
```

Restore a training checkpoint.
Restores this `Checkpoint` and any objects it depends on.
When executing eagerly, either assigns values immediately if variables to restore have been created already, or defers restoration until the variables are created. Dependencies added after this call will be matched if they have a corresponding object in the checkpoint (the restore request will queue in any trackable object waiting for the expected dependency to be added).
When graph building, restoration ops are added to the graph but not run immediately.

To ensure that loading is complete and no more assignments will take place, use the `assert_consumed()` method of the status object returned by `restore`:

```
checkpoint = tf.train.Checkpoint( ... )
checkpoint.restore(path).assert_consumed()
```

An exception will be raised if any Python objects in the dependency graph were not found in the checkpoint, or if any checkpointed values do not have a matching Python object.
When graph building, `assert_consumed()` indicates that all of the restore ops that will be created for this checkpoint have been created. They can be run via the `run_restore_ops()` method of the status object:

```
checkpoint.restore(path).assert_consumed().run_restore_ops()
```

If the checkpoint has not been consumed completely, then the list of restore ops will grow as more objects are added to the dependency graph.
Name-based `tf.compat.v1.train.Saver` checkpoints can be loaded using this method. Names are used to match variables. No restore ops are created/run until `run_restore_ops()` or `initialize_or_restore()` are called on the returned status object when graph building, but there is restore-on-creation when executing eagerly. Re-encode name-based checkpoints using`tf.train.Checkpoint.save` as soon as possible.

Args:

- **`save_path`**: The path to the checkpoint, as returned by `save` or`tf.train.latest_checkpoint`. If None (as when there is no latest checkpoint for `tf.train.latest_checkpoint` to return), returns an object which may run initializers for objects in the dependency graph. If the checkpoint was written by the name-based`tf.compat.v1.train.Saver`, names are used to match variables.

Returns:
A load status object, which can be used to make assertions about the status of a checkpoint restoration and run initialization/restore ops.
The returned status object has the following methods:

- `assert_consumed()`: Raises an exception if any variables/objects are unmatched: either checkpointed values which don't have a matching Python object or Python objects in the dependency graph with no values in the checkpoint. This method returns the status object, and so may be chained with `initialize_or_restore` or `run_restore_ops`.
- `assert_existing_objects_matched()`: Raises an exception if any existing Python objects in the dependency graph are unmatched. Unlike `assert_consumed`, this assertion will pass if values in the checkpoint have no corresponding Python objects. For example a `tf.keras.Layer` object which has not yet been built, and so has not created any variables, will pass this assertion but fail `assert_consumed`. Useful when loading part of a larger checkpoint into a new Python program, e.g. a training checkpoint with a `tf.compat.v1.train.Optimizer`was saved but only the state required for inference is being loaded. This method returns the status object, and so may be chained with `initialize_or_restore` or `run_restore_ops`.
- `assert_nontrivial_match()`: Asserts that something aside from the root object was matched. This is a very weak assertion, but is useful for sanity checking in library code where objects may exist in the checkpoint which haven't been created in Python and some Python objects may not have a checkpointed value.
- `expect_partial()`: Silence warnings about incomplete checkpoint restores. Warnings are otherwise printed for unused parts of the checkpoint file or object when the `Checkpoint` object is deleted (often at program shutdown).
- `initialize_or_restore(session=None)`: When graph building, runs variable initializers if `save_path` is `None`, but otherwise runs restore operations. If no `session` is explicitly specified, the default session is used. No effect when executing eagerly (variables are initialized or restored eagerly).

- `run_restore_ops(session=None)`: When graph building, runs restore operations. If no `session` is explicitly specified, the default session is used. No effect when executing eagerly (restore operations are run eagerly). May only be called when `save_path` is not `None`.

save

```
save(
    file_prefix,
    session=None
)
```

Saves a training checkpoint and provides basic checkpoint management.
The saved checkpoint includes variables created by this object and any trackable objects it depends on at the time `Checkpoint.save()` is called.
`save` is a basic convenience wrapper around the `write` method, sequentially numbering checkpoints using `save_counter` and updating the metadata used by `tf.train.latest_checkpoint`. More advanced checkpoint management, for example garbage collection and custom numbering, may be provided by other utilities which also wrap `write` (`tf.contrib.checkpoint.CheckpointManager`for example).

Args:

- **`file_prefix`**: A prefix to use for the checkpoint filenames (/path/to/directory/and_a_prefix). Names are generated based on this prefix and `Checkpoint.save_counter`.
- **`session`**: The session to evaluate variables in. Ignored when executing eagerly. If not provided when graph building, the default session is used.

Returns:

The full path to the checkpoint.

write

```
write(
    file_prefix,
    session=None
)
```

Writes a training checkpoint.
The checkpoint includes variables created by this object and any trackable objects it depends on at the time `Checkpoint.write()` is called.
`write` does not number checkpoints, increment `save_counter`, or update the metadata used by `tf.train.latest_checkpoint`. It is primarily intended for use by higher level checkpoint management utilities. `save` provides a very basic implementation of these features.

Args:

- **`file_prefix`**: A prefix to use for the checkpoint filenames (/path/to/directory/and_a_prefix).
- **`session`**: The session to evaluate variables in. Ignored when executing eagerly. If not provided when graph building, the default session is used.

Returns:

The full path to the checkpoint (i.e. `file_prefix`).

tf.compat.v1.train.checkpoint_exists

Checks whether a V1 or V2 checkpoint exists with the specified prefix. (deprecated)

```
tf.compat.v1.train.checkpoint_exists(checkpoint_prefix)
```

Defined in `python/training/checkpoint_management.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use standard file APIs to check for files with this prefix.

This is the recommended way to check if a checkpoint exists, since it takes into account the naming difference between V1 and V2 formats.

Args:

- **`checkpoint_prefix`**: the prefix of a V1 or V2 checkpoint, with V2 taking priority. Typically the result of `Saver.save()` or that of `tf.train.latest_checkpoint()`, regardless of sharded/non-sharded or V1/V2.

Returns:

A bool, true iff a checkpoint referred to by `checkpoint_prefix` exists.

tf.compat.v1.train.ChiefSessionCreator

- Contents
- Class ChiefSessionCreator
- __init__
- Methods
  - create_session

Class `ChiefSessionCreator`

Creates a tf.compat.v1.Session for a chief.

Inherits From: `SessionCreator`

Defined in `python/training/monitored_session.py`.

`__init__`

```
__init__(
    scaffold=None,
    master='',
    config=None,
    checkpoint_dir=None,
    checkpoint_filename_with_path=None
)
```

Initializes a chief session creator.

Args:

- **`scaffold`**: A `Scaffold` used for gathering or building supportive ops. If not specified a default one is created. It's used to finalize the graph.
- **`master`**: `String` representation of the TensorFlow master to use.
- **`config`**: `ConfigProto` proto used to configure the session.
- **`checkpoint_dir`**: A string. Optional path to a directory where to restore variables.
- **`checkpoint_filename_with_path`**: Full file name path to the checkpoint file.

Methods

`create_session`

```
create_session()
```

tf.compat.v1.train.cosine_decay

Applies cosine decay to the learning rate.

```
tf.compat.v1.train.cosine_decay(
    learning_rate,
```

```
    global_step,
    decay_steps,
    alpha=0.0,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.
See [Loshchilov & Hutter, ICLR2016], SGDR: Stochastic Gradient Descent with Warm Restarts. https://arxiv.org/abs/1608.03983
When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies a cosine 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:

```
global_step = min(global_step, decay_steps)
cosine_decay = 0.5 * (1 + cos(pi * global_step / decay_steps))
decayed = (1 - alpha) * cosine_decay + alpha
decayed_learning_rate = learning_rate * decayed
```

Example usage:

```
decay_steps = 1000
lr_decayed = cosine_decay(learning_rate, global_step, decay_steps)
```

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.
- **`decay_steps`**: A scalar `int32` or `int64` `Tensor` or a Python number. Number of steps to decay over.
- **`alpha`**: A scalar `float32` or `float64` Tensor or a Python number. Minimum learning rate value as a fraction of learning_rate.
- **`name`**: String. Optional name of the operation. Defaults to 'CosineDecay'.

Returns:

A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility

When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.cosine_decay_restarts

Applies cosine decay with restarts to the learning rate.

```
tf.compat.v1.train.cosine_decay_restarts(
    learning_rate,
    global_step,
    first_decay_steps,
    t_mul=2.0,
    m_mul=1.0,
    alpha=0.0,
```

```
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.
See [Loshchilov & Hutter, ICLR2016], SGDR: Stochastic Gradient Descent with Warm Restarts. https://arxiv.org/abs/1608.03983
When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies a cosine decay function with restarts 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 while taking into account possible warm restarts. The learning rate multiplier first decays from 1 to `alpha` for `first_decay_steps` steps. Then, a warm restart is performed. Each new warm restart runs for `t_mul` times more steps and with `m_mul` times smaller initial learning rate.

Example usage:

```
first_decay_steps = 1000
lr_decayed = cosine_decay_restarts(learning_rate, global_step,
                                   first_decay_steps)
```

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.
- **`first_decay_steps`**: A scalar `int32` or `int64` `Tensor` or a Python number. Number of steps to decay over.
- **`t_mul`**: A scalar `float32` or `float64` `Tensor` or a Python number. Used to derive the number of iterations in the i-th period
- **`m_mul`**: A scalar `float32` or `float64` `Tensor` or a Python number. Used to derive the initial learning rate of the i-th period:
- **`alpha`**: A scalar `float32` or `float64` Tensor or a Python number. Minimum learning rate value as a fraction of the learning_rate.
- **`name`**: String. Optional name of the operation. Defaults to 'SGDRDecay'.

Returns:
A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility
When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.create_global_step

Create global step tensor in graph.

```
tf.compat.v1.train.create_global_step(graph=None)
```

Defined in `python/training/training_util.py`.

Args:

- **`graph`**: The graph in which to create the global step tensor. If missing, use default graph.

Returns:
Global step tensor.

Raises:

- `ValueError`: if global step tensor is already defined.

- # tf.compat.v1.train.do_quantize_training_on_graphdef

- A general quantization scheme is being developed in `tf.contrib.quantize`. (deprecated)
- ```
  tf.compat.v1.train.do_quantize_training_on_graphdef(
      input_graph,
      num_bits
  )
  ```
- Defined in `python/pywrap_tensorflow_internal.py`.
- **Warning:** THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: GraphDef quantized training rewriter is deprecated in the long termConsider using that instead, though since it is in the tf.contrib namespace, it is notsubject to backward compatibility guarantees.

tf.compat.v1.train.exponential_decay

Applies exponential decay to the learning rate.

```
tf.compat.v1.train.exponential_decay(
    learning_rate,
    global_step,
    decay_steps,
    decay_rate,
    staircase=False,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

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.compat.v1.train.exponential_decay(starter_learning_rate,
global_step,
                                           100000, 0.96, staircase=True)
```
```

```
# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.compat.v1.train.GradientDescentOptimizer(learning_rate)
    .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. If `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.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility

When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.export_meta_graph

Returns `MetaGraphDef` proto.

```
tf.compat.v1.train.export_meta_graph(
    filename=None,
    meta_info_def=None,
    graph_def=None,
    saver_def=None,
    collection_list=None,
    as_text=False,
    graph=None,
    export_scope=None,
    clear_devices=False,
    clear_extraneous_savers=False,
    strip_default_attrs=False,
    save_debug_info=False,
    **kwargs
)
```

Defined in `python/training/saver.py`.

Optionally writes it to filename.

This function exports the graph, saver, and collection objects into `MetaGraphDef` protocol buffer with the intention 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.
- **`graph`**: The `Graph` to export. If `None`, use the default graph.
- **`export_scope`**: Optional `string`. Name scope under which to extract the subgraph. The scope name will be striped from the node definitions for easy import later into new name scopes. If `None`, the whole graph is exported. graph_def and export_scope cannot both be specified.
- **`clear_devices`**: Whether or not to clear the device field for an `Operation` or `Tensor` during export.
- **`clear_extraneous_savers`**: Remove any Saver-related information from the graph (both Save/Restore ops and SaverDefs) that are not associated with the provided SaverDef.
- **`strip_default_attrs`**: Boolean. If `True`, default-valued attributes will be removed from the NodeDefs. For a detailed guide, see [Stripping Default-Valued Attributes]().
- **`save_debug_info`**: If `True`, save the GraphDebugInfo to a separate file, which in the same directory of filename and with `_debug` added before the file extend.
- **`**kwargs`**: Optional keyed arguments.

Returns:

A `MetaGraphDef` proto.

Raises:

- **`ValueError`**: When the `GraphDef` is larger than 2GB.
- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility

Exporting/importing meta graphs is not supported unless both `graph_def` and `graph` are provided. No graph exists when eager execution is enabled.

tf.compat.v1.train.FtrlOptimizer

- **Contents**

Class `FtrlOptimizer`

Optimizer that implements the FTRL algorithm.

Inherits From: `Optimizer`

Defined in `python/training/ftrl.py`.

See this [paper](). This version has support for both online L2 (the L2 penalty given in the paper above) and shrinkage-type L2 (which is the addition of an L2 penalty to the loss function).

`__init__`

```
__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',
    accum_name=None,
    linear_name=None,
    l2_shrinkage_regularization_strength=0.0
)
```

Construct a new FTRL optimizer.

Args:

- **`learning_rate`**: A float value or a constant float `Tensor`.
- **`learning_rate_power`**: A float value, must be less or equal to zero. Controls how the learning rate decreases during training. Use zero for a fixed learning rate. See section 3.1 in the paper.
- **`initial_accumulator_value`**: The starting value for accumulators. Only zero or 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".
- **`accum_name`**: The suffix for the variable that keeps the gradient squared accumulator. If not present, defaults to name.
- **`linear_name`**: The suffix for the variable that keeps the linear gradient accumulator. If not present, defaults to name + "_1".
- **`l2_shrinkage_regularization_strength`**: A float value, must be greater than or equal to zero. This differs from L2 above in that the L2 above is a stabilization penalty, whereas this L2 shrinkage is a magnitude penalty. The FTRL formulation can be written as: w_{t+1} = argmin*w(\hat{g}*{1:t}w + L1||*w*||*_1 + L2*||w||_2^2), where \hat{g} = g + (2*L2_shrinkage*w), and g is the gradient of the loss function w.r.t. the weights w. Specifically, in the absence of L1 regularization, it is equivalent to the following update rule: w_{t+1} = w_t - lr_t / (1 + 2*L2*lr_t) * g_t - 2*L2_shrinkage*lr_t / (1 + 2*L2*lr_t) * w_t where lr_t is the learning rate at t. When input is sparse shrinkage will only happen on the active weights.

Raises:

- **`ValueError`**: If one of the arguments is invalid.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

compute_gradients

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.generate_checkpoint_state_proto

Generates a checkpoint state proto.

```
tf.compat.v1.train.generate_checkpoint_state_proto(
    save_dir,
    model_checkpoint_path,
    all_model_checkpoint_paths=None,
    all_model_checkpoint_timestamps=None,
    last_preserved_timestamp=None
)
```

Defined in `python/training/checkpoint_management.py`.

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.
- `all_model_checkpoint_timestamps`: A list of floats, indicating the number of seconds since the Epoch when each checkpoint was generated.
- `last_preserved_timestamp`: A float, indicating the number of seconds since the Epoch when the last preserved checkpoint was written, e.g. due to a `keep_checkpoint_every_n_hours`parameter (see `tf.contrib.checkpoint.CheckpointManager` for an implementation).

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.

Raises:

- `ValueError`: If `all_model_checkpoint_timestamps` was provided but its length does not match `all_model_checkpoint_paths`.

tf.compat.v1.train.get_checkpoint_mtimes

Returns the mtimes (modification timestamps) of the checkpoints. (deprecated)

```
tf.compat.v1.train.get_checkpoint_mtimes(checkpoint_prefixes)
```

Defined in `python/training/checkpoint_management.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use standard file utilities to get mtimes.

Globs for the checkpoints pointed to by `checkpoint_prefixes`. If the files exist, collect their mtime. Both V2 and V1 checkpoints are considered, in that priority.

This is the recommended way to get the mtimes, since it takes into account the naming difference between V1 and V2 formats.

Note: If not all checkpoints exist, the length of the returned mtimes list will be smaller than the length of `checkpoint_prefixes` list, so mapping checkpoints to corresponding mtimes will not be possible.

Args:

- `checkpoint_prefixes`: a list of checkpoint paths, typically the results of `Saver.save()` or those of `tf.train.latest_checkpoint()`, regardless of sharded/non-sharded or V1/V2.

Returns:
A list of mtimes (in microseconds) of the found checkpoints.

tf.compat.v1.train.get_global_step

- **Contents**
- Used in the guide:

Get the global step tensor.

```
tf.compat.v1.train.get_global_step(graph=None)
```

Defined in `python/training/training_util.py`.

Used in the guide:

- Training checkpoints

The global step tensor must be an integer variable. We first try to find it in the collection `GLOBAL_STEP`, or by name `global_step:0`.

Args:

- `graph`: The graph to find the global step in. If missing, use default graph.

Returns:
The global step variable, or `None` if none was found.

Raises:

- **`TypeError`**: If the global step tensor has a non-integer type, or if it is not a `Variable`.

tf.compat.v1.train.get_or_create_global_step

- **Contents**
- Used in the guide:
- Used in the tutorials:

Returns and create (if necessary) the global step tensor.

```
tf.compat.v1.train.get_or_create_global_step(graph=None)
```

Defined in `python/training/training_util.py`.

Used in the guide:

- Convert Your Existing Code to TensorFlow 2.0

Used in the tutorials:

- Multi-worker Training with Estimator

Args:

- **`graph`**: The graph in which to create the global step tensor. If missing, use default graph.

Returns:
The global step tensor.

tf.compat.v1.train.global_step

Small helper to get the global step.

```
tf.compat.v1.train.global_step(
    sess,
    global_step_tensor
)
```

Defined in `python/training/training_util.py`.

```
# Create a variable to hold the global_step.
global_step_tensor = tf.Variable(10, trainable=False, name='global_step')
# Create a session.
sess = tf.compat.v1.Session()
# Initialize the variable
sess.run(global_step_tensor.initializer)
# Get the variable value.
print('global_step: %s' % tf.compat.v1.train.global_step(sess,
global_step_tensor))

global_step: 10
```

Args:

- **`sess`**: A TensorFlow `Session` object.
- **`global_step_tensor`**: `Tensor` or the `name` of the operation that contains the global step.

Returns:
The global step value.

tf.compat.v1.train.GradientDescentOptimizer

- **Contents**
- Class GradientDescentOptimizer
  - Used in the tutorials:
- __init__
- Methods

Class `GradientDescentOptimizer`

Optimizer that implements the gradient descent algorithm.
Inherits From: `Optimizer`
Defined in `python/training/gradient_descent.py`.

Used in the tutorials:

- Multi-worker Training with Estimator

`__init__`

```
__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".

Eager Compatibility
When eager execution is enabled, `learning_rate` can be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

compute_gradients

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:
- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.import_meta_graph

Recreates a Graph saved in a `MetaGraphDef` proto.

```
tf.compat.v1.train.import_meta_graph(
    meta_graph_or_file,
    clear_devices=False,
    import_scope=None,
    **kwargs
)
```

Defined in `python/training/saver.py`.
This function takes a `MetaGraphDef` protocol buffer as input. If the argument is a file containing a `MetaGraphDef` protocol buffer , it constructs a protocol buffer from the file content. The function then adds all the nodes from the `graph_def` field to the current graph, recreates all the collections, and returns a saver constructed from the `saver_def` field.
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.

```
...
# Create a saver.
saver = tf.compat.v1.train.Saver(...variables...)
# Remember the training_op we want to run by adding it to a collection.
tf.compat.v1.add_to_collection('train_op', train_op)
sess = tf.compat.v1.Session()
for step in xrange(1000000):
    sess.run(train_op)
    if step % 1000 == 0:
        # Saves checkpoint, which by default also exports a meta_graph
        # named 'my-model-global_step.meta'.
        saver.save(sess, 'my-model', global_step=step)
```

Later we can continue training from this saved `meta_graph` without building the model from scratch.

```
with tf.compat.v1.Session() as sess:
  new_saver =
  tf.compat.v1.train.import_meta_graph('my-save-dir/my-model-10000.meta')
  new_saver.restore(sess, 'my-save-dir/my-model-10000')
  # tf.compat.v1.get_collection() returns a list. In this example we only want
  # the first one.
  train_op = tf.compat.v1.get_collection('train_op')[0]
  for step in xrange(1000000):
    sess.run(train_op)
```

NOTE: Restarting training from saved `meta_graph` only works if the device assignments have not changed.

Example 2:

Variables, placeholders, and independent operations can also be stored, as shown in the following example.

```
# Saving contents and operations.
v1 = tf.compat.v1.placeholder(tf.float32, name="v1")
v2 = tf.compat.v1.placeholder(tf.float32, name="v2")
v3 = tf.mul(v1, v2)
vx = tf.Variable(10.0, name="vx")
v4 = tf.add(v3, vx, name="v4")
saver = tf.compat.v1.train.Saver([vx])
sess = tf.compat.v1.Session()
sess.run(tf.compat.v1.initialize_all_variables())
sess.run(vx.assign(tf.add(vx, vx)))
result = sess.run(v4, feed_dict={v1:12.0, v2:3.3})
print(result)
saver.save(sess, "./model_ex1")
```

Later this model can be restored and contents loaded.

```
# Restoring variables and running operations.
saver = tf.compat.v1.train.import_meta_graph("./model_ex1.meta")
sess = tf.compat.v1.Session()
```

```
saver.restore(sess, "./model_ex1")
result = sess.run("v4:0", feed_dict={"v1:0": 12.0, "v2:0": 3.3})
print(result)
```

Args:

- **`meta_graph_or_file`**: `MetaGraphDef` protocol buffer or filename (including the path) containing a `MetaGraphDef`.
- **`clear_devices`**: Whether or not to clear the device field for an `Operation` or `Tensor` during import.
- **`import_scope`**: Optional `string`. Name scope to add. Only used when initializing from protocol buffer.
- **`**kwargs`**: Optional keyed arguments.

Returns:

A saver constructed from `saver_def` in `MetaGraphDef` or None.
A None value is returned if no variables exist in the `MetaGraphDef` (i.e., there are no variables to restore).

Raises:

- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility

Exporting/importing meta graphs is not supported. No graph exists when eager execution is enabled.

tf.compat.v1.train.init_from_checkpoint

Replaces `tf.Variable` initializers so they load from a checkpoint file.

```
tf.compat.v1.train.init_from_checkpoint(
    ckpt_dir_or_file,
    assignment_map
)
```

Defined in `python/training/checkpoint_utils.py`.
Values are not loaded immediately, but when the initializer is run (typically by running a `tf.compat.v1.global_variables_initializer` op).
Note: This overrides default initialization ops of specified variables and redefines dtype.

Assignment map supports following syntax:

- `'checkpoint_scope_name/': 'scope_name/'` - will load all variables in current `scope_name`from `checkpoint_scope_name` with matching tensor names.
- `'checkpoint_scope_name/some_other_variable': 'scope_name/variable_name'` - will initialize `scope_name/variable_name` variable from `checkpoint_scope_name/some_other_variable`.
- `'scope_variable_name': variable` - will initialize given `tf.Variable` object with tensor 'scope_variable_name' from the checkpoint.
- `'scope_variable_name': list(variable)` - will initialize list of partitioned variables with tensor 'scope_variable_name' from the checkpoint.
- `'/': 'scope_name/'` - will load all variables in current `scope_name` from checkpoint's root (e.g. no scope).

Supports loading into partitioned variables, which are represented as `'<variable>/part_<part #>'`.

Example:

```
# Say, '/tmp/model.ckpt' has the following tensors:
#  -- name='old_scope_1/var1', shape=[20, 2]
```

```
#  -- name='old_scope_1/var2', shape=[50, 4]
#  -- name='old_scope_2/var3', shape=[100, 100]

# Create new model's variables
with tf.compat.v1.variable_scope('new_scope_1'):
  var1 = tf.compat.v1.get_variable('var1', shape=[20, 2],
                         initializer=tf.compat.v1.zeros_initializer())
with tf.compat.v1.variable_scope('new_scope_2'):
  var2 = tf.compat.v1.get_variable('var2', shape=[50, 4],
                         initializer=tf.compat.v1.zeros_initializer())
  # Partition into 5 variables along the first axis.
  var3 = tf.compat.v1.get_variable(name='var3', shape=[100, 100],
                         initializer=tf.compat.v1.zeros_initializer(),
                         partitioner=lambda shape, dtype: [5, 1])

# Initialize all variables in `new_scope_1` from `old_scope_1`.
init_from_checkpoint('/tmp/model.ckpt', {'old_scope_1/': 'new_scope_1'})

# Use names to specify which variables to initialize from checkpoint.
init_from_checkpoint('/tmp/model.ckpt',
                     {'old_scope_1/var1': 'new_scope_1/var1',
                      'old_scope_1/var2': 'new_scope_2/var2'})

# Or use tf.Variable objects to identify what to initialize.
init_from_checkpoint('/tmp/model.ckpt',
                     {'old_scope_1/var1': var1,
                      'old_scope_1/var2': var2})

# Initialize partitioned variables using variable's name
init_from_checkpoint('/tmp/model.ckpt',
                     {'old_scope_2/var3': 'new_scope_2/var3'})

# Or specify the list of tf.Variable objects.
init_from_checkpoint('/tmp/model.ckpt',
                     {'old_scope_2/var3': var3._get_variable_list()})
```

Args:

- **`ckpt_dir_or_file`**: Directory with checkpoints file or path to checkpoint.
- **`assignment_map`**: Dict, where keys are names of the variables in the checkpoint and values are current variables or names of current variables (in default graph).

Raises:

- **`ValueError`**: If missing variables in current graph, or if missing checkpoints or tensors in checkpoints.

tf.compat.v1.train.input_producer

Output the rows of `input_tensor` to a queue for an input pipeline. (deprecated)

```
tf.compat.v1.train.input_producer(
    input_tensor,
    element_shape=None,
    num_epochs=None,
    shuffle=True,
    seed=None,
    capacity=32,
    shared_name=None,
    summary_name=None,
    name=None,
    cancel_op=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.from_tensor_slices(input_tensor).shuffle(tf.shape(input_tensor, out_type=tf.int64)[0]).repeat(num_epochs)`. If `shuffle=False`, omit the `.shuffle(...)`.**Note:** if `num_epochs` is not `None`, this function creates local counter `epochs`. Use `local_variables_initializer()` to initialize local variables.

Args:

- `input_tensor`: A tensor with the rows to produce. Must be at least one-dimensional. Must either have a fully-defined shape, or `element_shape` must be defined.
- `element_shape`: (Optional.) A `TensorShape` representing the shape of a row of `input_tensor`, if it cannot be inferred.
- `num_epochs`: (Optional.) An integer. If specified `input_producer` produces each row of `input_tensor` `num_epochs` times before generating an `OutOfRange` error. If not specified, `input_producer` can cycle through the rows of `input_tensor` an unlimited number of times.
- `shuffle`: (Optional.) A boolean. If true, the rows are randomly shuffled within each epoch.
- `seed`: (Optional.) An integer. The seed to use if `shuffle` is true.
- `capacity`: (Optional.) The capacity of the queue to be used for buffering the input.
- `shared_name`: (Optional.) If set, this queue will be shared under the given name across multiple sessions.
- `summary_name`: (Optional.) If set, a scalar summary for the current queue size will be generated, using this name as part of the tag.
- `name`: (Optional.) A name for queue.
- `cancel_op`: (Optional.) Cancel op for the queue

Returns:

A queue with the output rows. A `QueueRunner` for the queue is added to the current `QUEUE_RUNNER`collection of the current graph.

Raises:

- `ValueError`: If the shape of the input cannot be inferred from the arguments.
- `RuntimeError`: If called with eager execution enabled.

Eager Compatibility

Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.inverse_time_decay

Applies inverse time decay to the initial learning rate.

```
tf.compat.v1.train.inverse_time_decay(
    learning_rate,
    global_step,
    decay_steps,
    decay_rate,
    staircase=False,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies an inverse decay function to a provided initial learning rate. It requires an `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 / (1 + decay_rate * global_step /
decay_step)
```

or, if `staircase` is `True`, as:

```
decayed_learning_rate = learning_rate / (1 + decay_rate * floor(global_step /
decay_step))
```

Example: decay 1/t with a rate of 0.5:

```
...
global_step = tf.Variable(0, trainable=False)
learning_rate = 0.1
decay_steps = 1.0
decay_rate = 0.5
learning_rate = tf.compat.v1.train.inverse_time_decay(learning_rate,
global_step,
decay_steps, decay_rate)

# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.compat.v1.train.GradientDescentOptimizer(learning_rate)
    .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 Python number. Global step to use for the decay computation. Must not be negative.
- **`decay_steps`**: How often to apply decay.
- **`decay_rate`**: A Python number. The decay rate.
- **`staircase`**: Whether to apply decay in a discrete staircase, as opposed to continuous, fashion.
- **`name`**: String. Optional name of the operation. Defaults to 'InverseTimeDecay'.

Returns:
A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility
When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.limit_epochs

Returns tensor `num_epochs` times and then raises an `OutOfRange` error. (deprecated)

```
tf.compat.v1.train.limit_epochs(
    tensor,
    num_epochs=None,
    name=None
)
```

Defined in `python/training/input.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by **`tf.data`**. Use **`tf.data.Dataset.from_tensors(tensor).repeat(num_epochs)`**.**Note:** creates local counter **`epochs`**. Use **`local_variables_initializer()`** to initialize local variables.

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.compat.v1.train.linear_cosine_decay

Applies linear cosine decay to the learning rate.

```
tf.compat.v1.train.linear_cosine_decay(
    learning_rate,
    global_step,
    decay_steps,
    num_periods=0.5,
    alpha=0.0,
    beta=0.001,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.
See [Bello et al., ICML2017] Neural Optimizer Search with RL. https://arxiv.org/abs/1709.07417
For the idea of warm starts here controlled by `num_periods`, see [Loshchilov & Hutter, ICLR2016] SGDR: Stochastic Gradient Descent with Warm Restarts. https://arxiv.org/abs/1608.03983

Note that linear cosine decay is more aggressive than cosine decay and larger initial learning rates can typically be used.
When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies a linear cosine 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:

```
global_step = min(global_step, decay_steps)
linear_decay = (decay_steps - global_step) / decay_steps)
cosine_decay = 0.5 * (
    1 + cos(pi * 2 * num_periods * global_step / decay_steps))
decayed = (alpha + linear_decay) * cosine_decay + beta
decayed_learning_rate = learning_rate * decayed
```

Example usage:

```
decay_steps = 1000
lr_decayed = linear_cosine_decay(learning_rate, global_step, decay_steps)
```

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.
- **`decay_steps`**: A scalar `int32` or `int64` `Tensor` or a Python number. Number of steps to decay over.
- **`num_periods`**: Number of periods in the cosine part of the decay. See computation above.
- **`alpha`**: See computation above.
- **`beta`**: See computation above.
- **`name`**: String. Optional name of the operation. Defaults to 'LinearCosineDecay'.

Returns:
A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility
When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.LooperThread

- **Contents**
- Class LooperThread
- __init__
- Properties
  - daemon

Class `LooperThread`

A thread that runs code repeatedly, optionally on a timer.
Defined in [`python/training/coordinator.py`](python/training/coordinator.py).
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.

`__init__`

```
__init__(
    coord,
    timer_interval_secs,
    target=None,
    args=None,
    kwargs=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.
- **`kwargs`**: Optional keyword arguments to pass to `target` when calling it.

Raises:

- **`ValueError`**: If one of the arguments is invalid.

Properties

`daemon`

A boolean value indicating whether this thread is a daemon thread.
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.

`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.

`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.

Methods

`getName`

```
getName()
```

isAlive

```
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.

isDaemon

```
isDaemon()
```

is_alive

```
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.

join

```
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.

loop

```
@staticmethod
loop(
    coord,
    timer_interval_secs,
    target,
    args=None,
    kwargs=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.
- **`kwargs`**: Optional keyword arguments to pass to `target` when calling it.

Returns:
The started thread.

run

```
run()
```

run_loop

```
run_loop()
```

Called at 'timer_interval_secs' boundaries.

setDaemon

```
setDaemon(daemonic)
```

setName

```
setName(name)
```

start

```
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.

start_loop

```
start_loop()
```

Called when the thread starts.

stop_loop

```
stop_loop()
```

Called when the thread stops.

tf.compat.v1.train.maybe_batch

Conditionally creates batches of tensors based on `keep_input`. (deprecated)

```
tf.compat.v1.train.maybe_batch(
    tensors,
    keep_input,
    batch_size,
    num_threads=1,
    capacity=32,
    enqueue_many=False,
    shapes=None,
```

```
    dynamic_pad=False,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`.
Use `tf.data.Dataset.filter(...).batch(batch_size)` (or `padded_batch(...)` if `dynamic_pad=True`).
See docstring in `batch` for more details.

Args:

- **`tensors`**: The list or dictionary of tensors to enqueue.
- **`keep_input`**: A `bool` Tensor. This tensor controls whether the input is added to the queue or not. If it is a scalar and evaluates `True`, then `tensors` are all added to the queue. If it is a vector and `enqueue_many` is `True`, then each example is added to the queue only if the corresponding value in `keep_input` is `True`. This tensor essentially acts as a filtering mechanism.
- **`batch_size`**: The new batch size pulled from the queue.
- **`num_threads`**: The number of threads enqueuing `tensors`. The batching will be nondeterministic if `num_threads > 1`.
- **`capacity`**: An integer. The maximum number of elements in the queue.
- **`enqueue_many`**: Whether each tensor in `tensors` is a single example.
- **`shapes`**: (Optional) The shapes for each example. Defaults to the inferred shapes for `tensors`.
- **`dynamic_pad`**: Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (Optional). If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same types as `tensors`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

tf.compat.v1.train.maybe_batch_join

Runs a list of tensors to conditionally fill a queue to create batches. (deprecated)

```
tf.compat.v1.train.maybe_batch_join(
    tensors_list,
    keep_input,
    batch_size,
    capacity=32,
    enqueue_many=False,
    shapes=None,
    dynamic_pad=False,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
```

```
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`.

Use `tf.data.Dataset.interleave(...).filter(...).batch(batch_size)` (or `padded_batch(...)` if `dynamic_pad=True`).

See docstring in `batch_join` for more details.

Args:

- `tensors_list`: A list of tuples or dictionaries of tensors to enqueue.
- `keep_input`: A `bool` Tensor. This tensor controls whether the input is added to the queue or not. If it is a scalar and evaluates `True`, then `tensors` are all added to the queue. If it is a vector and `enqueue_many` is `True`, then each example is added to the queue only if the corresponding value in `keep_input` is `True`. This tensor essentially acts as a filtering mechanism.
- `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]`.
- `dynamic_pad`: Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- `allow_smaller_final_batch`: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- `shared_name`: (Optional) If set, this queue will be shared under the given name across multiple sessions.
- `name`: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- `ValueError`: If the `shapes` are not specified, and cannot be inferred from the elements of `tensor_list_list`.

tf.compat.v1.train.maybe_shuffle_batch

Creates batches by randomly shuffling conditionally-enqueued tensors. (deprecated)

```
tf.compat.v1.train.maybe_shuffle_batch(
    tensors,
    batch_size,
    capacity,
    min_after_dequeue,
    keep_input,
    num_threads=1,
    seed=None,
    enqueue_many=False,
    shapes=None,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
```

```
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`.

Use `tf.data.Dataset.filter(...).shuffle(min_after_dequeue).batch(batch_size)`.

See docstring in `shuffle_batch` for more details.

Args:

- **`tensors`**: The list or dictionary 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.
- **`keep_input`**: A `bool` Tensor. This tensor controls whether the input is added to the queue or not. If it is a scalar and evaluates `True`, then `tensors` are all added to the queue. If it is a vector and `enqueue_many` is `True`, then each example is added to the queue only if the corresponding value in `keep_input` is `True`. This tensor essentially acts as a filtering mechanism.
- **`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`.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (Optional) If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the types as `tensors`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

Eager Compatibility

Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.maybe_shuffle_batch_join

Create batches by randomly shuffling conditionally-enqueued tensors. (deprecated)

```
tf.compat.v1.train.maybe_shuffle_batch_join(
    tensors_list,
    batch_size,
    capacity,
    min_after_dequeue,
    keep_input,
    seed=None,
    enqueue_many=False,
    shapes=None,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
```

```
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`.

Use `tf.data.Dataset.interleave(...).filter(...).shuffle(min_after_dequeue).batch(batch_size)`.

See docstring in `shuffle_batch_join` for more details.

Args:

- **`tensors_list`**: A list of tuples or dictionaries 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.
- **`keep_input`**: A `bool` Tensor. This tensor controls whether the input is added to the queue or not. If it is a scalar and evaluates `True`, then `tensors` are all added to the queue. If it is a vector and `enqueue_many` is `True`, then each example is added to the queue only if the corresponding value in `keep_input` is `True`. This tensor essentially acts as a filtering mechanism.
- **`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 `tensors_list[i]`.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (optional). If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors_list`.

Eager Compatibility

Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.MomentumOptimizer

- **Contents**
- Class MomentumOptimizer
- __init__
- Methods
  - apply_gradients

Class `MomentumOptimizer`

Optimizer that implements the Momentum algorithm.

Inherits From: `Optimizer`

Defined in `python/training/momentum.py`.

Computes (if `use_nesterov = False`):

```
accumulation = momentum * accumulation + gradient
variable -= learning_rate * accumulation
```

Note that in the dense version of this algorithm, `accumulation` is updated and applied regardless of a gradient's value, whereas the sparse version (when the gradient is an `IndexedSlices`, typically because of `tf.gather` or an embedding) only updates variable slices and corresponding `accumulation` terms when that part of the variable was used in the forward pass.

`__init__`

```
__init__(
    learning_rate,
    momentum,
    use_locking=False,
    name='Momentum',
    use_nesterov=False
)
```

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".
- **`use_nesterov`**: If `True` use Nesterov Momentum. See [Sutskever et al., 2013](). This implementation always computes gradients at the value of the variable(s) passed to the optimizer. Using Nesterov Momentum makes the variable(s) track the values called `theta_t + mu*v_t` in the paper. This implementation is an approximation of the original formula, valid for high values of momentum. It will compute the "adjusted gradient" in NAG by assuming that the new gradient will be estimated by the current average gradient plus the product of momentum and the change in the average gradient.

Eager Compatibility

When eager execution is enabled, `learning_rate` and `momentum` can each be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:
- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

`get_name`

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.MonitoredSession

- **Contents**
- Class MonitoredSession
- __init__
- Child Classes
- Properties

Class `MonitoredSession`

Session-like object that handles initialization, recovery and hooks.
Defined in `python/training/monitored_session.py`.

Example usage:

```
saver_hook = CheckpointSaverHook(...)
summary_hook = SummarySaverHook(...)
with MonitoredSession(session_creator=ChiefSessionCreator(...),
                      hooks=[saver_hook, summary_hook]) as sess:
  while not sess.should_stop():
    sess.run(train_op)
```

Initialization: At creation time the monitored session does following things in given order:

- calls `hook.begin()` for each given hook
- finalizes the graph via `scaffold.finalize()`
- create session
- initializes the model via initialization ops provided by `Scaffold`
- restores variables if a checkpoint exists
- launches queue runners
- calls `hook.after_create_session()`

Run: When `run()` is called, the monitored session does following things:

- calls `hook.before_run()`
- calls TensorFlow `session.run()` with merged fetches and feed_dict
- calls `hook.after_run()`
- returns result of `session.run()` asked by user
- if `AbortedError` or `UnavailableError` occurs, it recovers or reinitializes the session before executing the run() call again

Exit: At the `close()`, the monitored session does following things in order:

- calls `hook.end()`
- closes the queue runners and the session
- suppresses `OutOfRange` error which indicates that all inputs have been processed if the monitored_session is used as a context

How to set `tf.compat.v1.Session` arguments:

- In most cases you can set session arguments as follows:

```
MonitoredSession(
  session_creator=ChiefSessionCreator(master=..., config=...))
```

- In distributed setting for a non-chief worker, you can use following:

```
MonitoredSession(
  session_creator=WorkerSessionCreator(master=..., config=...))
```

See `MonitoredTrainingSession` for an example usage based on chief or worker.
Note: This is not a **`tf.compat.v1.Session`**. For example, it cannot do following:

- it cannot be set as default session.
- it cannot be sent to saver.save.
- it cannot be sent to tf.train.start_queue_runners.

Args:

- **`session_creator`**: A factory object to create session. Typically a `ChiefSessionCreator`which is the default one.
- **`hooks`**: An iterable of `SessionRunHook' objects.

Returns:
A MonitoredSession object.

`__init__`

```
__init__(
    session_creator=None,
    hooks=None,
    stop_grace_period_secs=120
)
```

Child Classes

`class StepContext`

Properties

`graph`

The graph that was launched in this session.

Methods

`__enter__`

```
__enter__()
```

`__exit__`

```
__exit__(
    exception_type,
    exception_value,
    traceback
)
```

`close`

```
close()
```

`run`

```
run(
    fetches,
    feed_dict=None,
    options=None,
    run_metadata=None
)
```

Run ops in the monitored session.
This method is completely compatible with the `tf.Session.run()` method.

Args:

- **`fetches`**: Same as `tf.Session.run()`.
- **`feed_dict`**: Same as `tf.Session.run()`.
- **`options`**: Same as `tf.Session.run()`.
- **`run_metadata`**: Same as `tf.Session.run()`.

Returns:
Same as `tf.Session.run()`.

`run_step_fn`

```
run_step_fn(step_fn)
```

Run ops using a step function.

Args:

- **`step_fn`**: A function or a method with a single argument of type `StepContext`. The function may use methods of the argument to perform computations with access to a raw session. The returned value of the `step_fn` will be returned from `run_step_fn`, unless a stop is requested. In that case, the next `should_stop` call will return True. Example usage: ```python with tf.Graph().as_default(): c = tf.compat.v1.placeholder(dtypes.float32) v = tf.add(c, 4.0) w = tf.add(c, 0.5) def step_fn(step_context): a = step_context.session.run(fetches=v, feed_dict={c: 0.5}) if a <= 4.5: step_context.request_stop() return step_context.run_with_hooks(fetches=w, feed_dict={c: 0.1}) with tf.MonitoredSession() as session: while not session.should_stop(): a = session.run_step_fn(step_fn)

```
    ```  Hooks interact with the `run_with_hooks()` call inside the
    `step_fn` as they do with a `MonitoredSession.run` call.

```

Returns:
Returns the returned value of `step_fn`.

Raises:

- **`StopIteration`**: if `step_fn` has called `request_stop()`. It may be caught by `with tf.MonitoredSession()` to close the session.
- **`ValueError`**: if `step_fn` doesn't have a single argument called `step_context`. It may also optionally have `self` for cases when it belongs to an object.

`should_stop`

```
should_stop()
```

tf.compat.v1.train.MonitoredSession.StepContext

- **Contents**
- Class StepContext
  - Aliases:
- __init__
- Properties

Class `StepContext`

Control flow instrument for the `step_fn` from `run_step_fn()`.

Aliases:

- Class `tf.compat.v1.train.MonitoredSession.StepContext`
- Class `tf.compat.v1.train.SingularMonitoredSession.StepContext`

Defined in `python/training/monitored_session.py`.
Users of `step_fn` may perform `run()` calls without running hooks by accessing the `session`. A `run()` call with hooks may be performed using `run_with_hooks()`. Computation flow can be interrupted using `request_stop()`.

`__init__`

```
__init__(
    session,
    run_with_hooks_fn
)
```

Initializes the `step_context` argument for a `step_fn` invocation.

Args:

- **`session`**: An instance of `tf.compat.v1.Session`.
- **`run_with_hooks_fn`**: A function for running fetches and hooks.
```

Properties

`session`

Methods

`request_stop`

```
request_stop()
```

Exit the training loop by causing `should_stop()` to return `True`.
Causes `step_fn` to exit by raising an exception.

Raises:
StopIteration

`run_with_hooks`

```
run_with_hooks(
    *args,
    **kwargs
)
```

Same as `MonitoredSession.run`. Accepts the same arguments.

tf.compat.v1.train.MonitoredTrainingSession

Creates a `MonitoredSession` for training.

```
tf.compat.v1.train.MonitoredTrainingSession(
    master='',
    is_chief=True,
    checkpoint_dir=None,
    scaffold=None,
    hooks=None,
    chief_only_hooks=None,
    save_checkpoint_secs=USE_DEFAULT,
    save_summaries_steps=USE_DEFAULT,
    save_summaries_secs=USE_DEFAULT,
    config=None,
    stop_grace_period_secs=120,
    log_step_count_steps=100,
    max_wait_secs=7200,
    save_checkpoint_steps=USE_DEFAULT,
    summary_dir=None
)
```

Defined in `python/training/monitored_session.py`.
For a chief, this utility sets proper session initializer/restorer. It also creates hooks related to checkpoint and summary saving. For workers, this utility sets proper session creator which waits for the chief to initialize/restore. Please check `tf.compat.v1.train.MonitoredSession` for more information.

Args:

- **`master`**: `String` the TensorFlow master to use.

- **`is_chief`**: If `True`, it will take care of initialization and recovery the underlying TensorFlow session. If `False`, it will wait on a chief to initialize or recover the TensorFlow session.
- **`checkpoint_dir`**: A string. Optional path to a directory where to restore variables.
- **`scaffold`**: A `Scaffold` used for gathering or building supportive ops. If not specified, a default one is created. It's used to finalize the graph.
- **`hooks`**: Optional list of `SessionRunHook` objects.
- **`chief_only_hooks`**: list of `SessionRunHook` objects. Activate these hooks if`is_chief==True`, ignore otherwise.
- **`save_checkpoint_secs`**: The frequency, in seconds, that a checkpoint is saved using a default checkpoint saver. If both `save_checkpoint_steps` and `save_checkpoint_secs` are set to `None`, then the default checkpoint saver isn't used. If both are provided, then only `save_checkpoint_secs` is used. Default 600.
- **`save_summaries_steps`**: The frequency, in number of global steps, that the summaries are written to disk using a default summary saver. If both `save_summaries_steps` and `save_summaries_secs` are set to `None`, then the default summary saver isn't used. Default 100.
- **`save_summaries_secs`**: The frequency, in secs, that the summaries are written to disk using a default summary saver. If both `save_summaries_steps` and `save_summaries_secs` are set to `None`, then the default summary saver isn't used. Default not enabled.
- **`config`**: an instance of `tf.compat.v1.ConfigProto` proto used to configure the session. It's the `config` argument of constructor of `tf.compat.v1.Session`.
- **`stop_grace_period_secs`**: Number of seconds given to threads to stop after `close()` has been called.
- **`log_step_count_steps`**: The frequency, in number of global steps, that the global step/sec is logged.
- **`max_wait_secs`**: Maximum time workers should wait for the session to become available. This should be kept relatively short to help detect incorrect code, but sometimes may need to be increased if the chief takes a while to start up.
- **`save_checkpoint_steps`**: The frequency, in number of global steps, that a checkpoint is saved using a default checkpoint saver. If both `save_checkpoint_steps` and `save_checkpoint_secs` are set to `None`, then the default checkpoint saver isn't used. If both are provided, then only `save_checkpoint_secs` is used. Default not enabled.
- **`summary_dir`**: A string. Optional path to a directory where to save summaries. If None, checkpoint_dir is used instead.

Returns:

A `MonitoredSession` object.

tf.compat.v1.train.natural_exp_decay

Applies natural exponential decay to the initial learning rate.

```
tf.compat.v1.train.natural_exp_decay(
    learning_rate,
    global_step,
    decay_steps,
    decay_rate,
    staircase=False,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

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 an `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 * exp(-decay_rate * global_step /
decay_step)
```

or, if `staircase` is `True`, as:

```
decayed_learning_rate = learning_rate * exp(-decay_rate * floor(global_step /
decay_step))
```

Example: decay exponentially with a base of 0.96:

```
...
global_step = tf.Variable(0, trainable=False)
learning_rate = 0.1
decay_steps = 5
k = 0.5
learning_rate = tf.compat.v1.train.natural_exp_decay(learning_rate,
global_step,
                                           decay_steps, k)

# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.compat.v1.train.GradientDescentOptimizer(learning_rate)
    .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 Python number. Global step to use for the decay computation. Must not be negative.
- **`decay_steps`**: How often to apply decay.
- **`decay_rate`**: A Python number. The decay rate.
- **`staircase`**: Whether to apply decay in a discrete staircase, as opposed to continuous, fashion.
- **`name`**: String. Optional name of the operation. Defaults to 'ExponentialTimeDecay'.

Returns:
A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility
When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.NewCheckpointReader

```
tf.compat.v1.train.NewCheckpointReader(filepattern)
```

Defined in `python/pywrap_tensorflow_internal.py`.

tf.compat.v1.train.noisy_linear_cosine_decay

Applies noisy linear cosine decay to the learning rate.

```
tf.compat.v1.train.noisy_linear_cosine_decay(
    learning_rate,
    global_step,
    decay_steps,
    initial_variance=1.0,
    variance_decay=0.55,
    num_periods=0.5,
    alpha=0.0,
    beta=0.001,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

See [Bello et al., ICML2017] Neural Optimizer Search with RL. https://arxiv.org/abs/1709.07417

For the idea of warm starts here controlled by `num_periods`, see [Loshchilov & Hutter, ICLR2016] SGDR: Stochastic Gradient Descent with Warm Restarts. https://arxiv.org/abs/1608.03983

Note that linear cosine decay is more aggressive than cosine decay and larger initial learning rates can typically be used.

When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies a noisy linear cosine 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:

```
global_step = min(global_step, decay_steps)
linear_decay = (decay_steps - global_step) / decay_steps)
cosine_decay = 0.5 * (
    1 + cos(pi * 2 * num_periods * global_step / decay_steps))
decayed = (alpha + linear_decay + eps_t) * cosine_decay + beta
decayed_learning_rate = learning_rate * decayed
```

where eps_t is 0-centered gaussian noise with variance initial_variance / (1 + global_step) ** variance_decay

Example usage:

```
decay_steps = 1000
lr_decayed = noisy_linear_cosine_decay(
  learning_rate, global_step, decay_steps)
```

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.
- **`decay_steps`**: A scalar `int32` or `int64` `Tensor` or a Python number. Number of steps to decay over.
- **`initial_variance`**: initial variance for the noise. See computation above.
- **`variance_decay`**: decay for the noise's variance. See computation above.
- **`num_periods`**: Number of periods in the cosine part of the decay. See computation above.
- **`alpha`**: See computation above.

- `beta`: See computation above.
- `name`: String. Optional name of the operation. Defaults to 'NoisyLinearCosineDecay'.

Returns:

A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- `ValueError`: if `global_step` is not supplied.

Eager Compatibility

When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.Optimizer

- **Contents**

Class `Optimizer`

Base class for optimizers.

Defined in `python/training/optimizer.py`.

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)
```

Gating Gradients
Both `minimize()` and `compute_gradients()` accept a `gate_gradients` 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 *before* 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.

`__init__`

```
__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.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

`get_name`

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility

When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

variables

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:

A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.piecewise_constant

- **Contents**
- Aliases:

Piecewise constant from boundaries and interval values.

Aliases:

- `tf.compat.v1.train.piecewise_constant`
- `tf.compat.v1.train.piecewise_constant_decay`

```
tf.compat.v1.train.piecewise_constant(
    x,
    boundaries,
```

```
    values,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

Example: use a learning rate that's 1.0 for the first 100001 steps, 0.5 for the next 10000 steps, and 0.1 for any additional steps.

```
global_step = tf.Variable(0, trainable=False)
boundaries = [100000, 110000]
values = [1.0, 0.5, 0.1]
learning_rate = tf.compat.v1.train.piecewise_constant(global_step, boundaries,
values)

# Later, whenever we perform an optimization step, we increment global_step.
```

Args:

- **`x`**: A 0-D scalar `Tensor`. Must be one of the following types: `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`.
- **`boundaries`**: A list of `Tensor`s or `int`s or `float`s with strictly increasing entries, and with all elements having the same type as `x`.
- **`values`**: A list of `Tensor`s or `float`s or `int`s that specifies the values for the intervals defined by `boundaries`. It should have one more element than `boundaries`, and all elements should have the same type.
- **`name`**: A string. Optional name of the operation. Defaults to 'PiecewiseConstant'.

Returns:

A 0-D Tensor. Its value is `values[0]` when `x <= boundaries[0]`, `values[1]` when `x > boundaries[0]` and `x <= boundaries[1]`, ..., and values[-1] when `x > boundaries[-1]`.

Raises:

- **`ValueError`**: if types of `x` and `boundaries` do not match, or types of all `values` do not match or the number of elements in the lists does not match.

Eager Compatibility

When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.polynomial_decay

Applies a polynomial decay to the learning rate.

```
tf.compat.v1.train.polynomial_decay(
    learning_rate,
    global_step,
    decay_steps,
    end_learning_rate=0.0001,
    power=1.0,
    cycle=False,
    name=None
)
```

Defined in `python/training/learning_rate_decay.py`.

It is commonly observed that a monotonically decreasing learning rate, whose degree of change is carefully chosen, results in a better performing model. This function applies a polynomial decay function to a provided initial `learning_rate` to reach an `end_learning_rate` in the given `decay_steps`.
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:

```
global_step = min(global_step, decay_steps)
decayed_learning_rate = (learning_rate - end_learning_rate) *
                        (1 - global_step / decay_steps) ^ (power) +
                        end_learning_rate
```

If `cycle` is True then a multiple of `decay_steps` is used, the first one that is bigger than `global_steps`.

```
decay_steps = decay_steps * ceil(global_step / decay_steps)
decayed_learning_rate = (learning_rate - end_learning_rate) *
                        (1 - global_step / decay_steps) ^ (power) +
                        end_learning_rate
```

Example: decay from 0.1 to 0.01 in 10000 steps using sqrt (i.e. power=0.5):

```
...
global_step = tf.Variable(0, trainable=False)
starter_learning_rate = 0.1
end_learning_rate = 0.01
decay_steps = 10000
learning_rate = tf.compat.v1.train.polynomial_decay(starter_learning_rate,
global_step,
                                          decay_steps, end_learning_rate,
                                          power=0.5)
# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.compat.v1.train.GradientDescentOptimizer(learning_rate)
    .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.
- **`end_learning_rate`**: A scalar `float32` or `float64` `Tensor` or a Python number. The minimal end learning rate.
- **`power`**: A scalar `float32` or `float64` `Tensor` or a Python number. The power of the polynomial. Defaults to linear, 1.0.
- **`cycle`**: A boolean, whether or not it should cycle beyond decay_steps.
- **`name`**: String. Optional name of the operation. Defaults to 'PolynomialDecay'.

Returns:
A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Raises:

- **`ValueError`**: if `global_step` is not supplied.

Eager Compatibility
When eager execution is enabled, this function returns a function which in turn returns the decayed learning rate Tensor. This can be useful for changing the learning rate value across different invocations of optimizer functions.

tf.compat.v1.train.ProximalAdagradOptimizer

Class `ProximalAdagradOptimizer`

Optimizer that implements the Proximal Adagrad algorithm.
Inherits From: `Optimizer`
Defined in `python/training/proximal_adagrad.py`.
See this paper.

`__init__`

```
__init__(
    learning_rate,
    initial_accumulator_value=0.1,
    l1_regularization_strength=0.0,
    l2_regularization_strength=0.0,
    use_locking=False,
    name='ProximalAdagrad'
)
```

Construct a new ProximalAdagrad 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.
- **`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 "Adagrad".

Raises:

- **`ValueError`**: If the `initial_accumulator_value` is invalid.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

compute_gradients

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:
A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.

- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility
When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

`get_name`

```
get_name()
```

`get_slot`

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility

When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.

Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:

A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.ProximalGradientDescentOptimizer

- **Contents**
- Class ProximalGradientDescentOptimizer
- __init__
- Methods
  - apply_gradients

Class `ProximalGradientDescentOptimizer`

Optimizer that implements the proximal gradient descent algorithm.

Inherits From: `Optimizer`
Defined in `python/training/proximal_gradient_descent.py`.
See this paper.

`__init__`

```
__init__(
    learning_rate,
    l1_regularization_strength=0.0,
    l2_regularization_strength=0.0,
    use_locking=False,
    name='ProximalGradientDescent'
)
```

Construct a new proximal gradient descent optimizer.

Args:

- **`learning_rate`**: A Tensor or a floating point value. The learning rate to use.
- **`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 "GradientDescent".

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
```

```
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility

When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

get_slot

```
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.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.
See `get_slot()`.

Returns:
A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility
When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of

the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

variables

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.QueueRunner

- **Contents**
- Class QueueRunner
  - Aliases:
- __init__
- Properties

Class `QueueRunner`

Holds a list of enqueue operations for a queue, each to be run in a thread.

Aliases:

- Class `tf.compat.v1.train.QueueRunner`
- Class `tf.compat.v1.train.queue_runner.QueueRunner`

Defined in `python/training/queue_runner_impl.py`.
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.

Eager Compatibility
QueueRunners are not compatible with eager execution. Instead, please use `tf.data` to get data into your model.

__init__

```
__init__(
    queue=None,
    enqueue_ops=None,
    close_op=None,
    cancel_op=None,
    queue_closed_exception_types=None,
    queue_runner_def=None,
    import_scope=None
```

```
)
```

Create a QueueRunner. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: To construct input pipelines, use the `tf.data` module.

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_closed_exception_types`**: Optional tuple of Exception types that indicate that the queue has been closed when raised during an enqueue operation. Defaults to `(tf.errors.OutOfRangeError,)`. Another common case includes `(tf.errors.OutOfRangeError, tf.errors.CancelledError)`, when some of the enqueue ops may dequeue from other Queues.
- **`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.
- **`import_scope`**: Optional `string`. Name scope to add. Only used when initializing from protocol buffer.

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`.
- **`RuntimeError`**: If eager execution is enabled.

Properties

`cancel_op`

`close_op`

`enqueue_ops`

`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.)

`name`

The string name of the underlying Queue.

`queue`

`queue_closed_exception_types`

Methods

`create_threads`

```
create_threads(
    sess,
    coord=None,
    daemon=False,
    start=False
)
```

Create threads to run the enqueue ops for the given session.
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.
If previously created threads for the given session are still running, no new threads will be created.

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.

`from_proto`

```
@staticmethod
from_proto(
    queue_runner_def,
    import_scope=None
)
```

Returns a `QueueRunner` object created from `queue_runner_def`.

`to_proto`

```
to_proto(export_scope=None)
```

Converts this `QueueRunner` to a `QueueRunnerDef` protocol buffer.

Args:

- **`export_scope`**: Optional `string`. Name scope to remove.

Returns:
A `QueueRunnerDef` protocol buffer, or `None` if the `Variable` is not in the specified name scope.

tf.compat.v1.train.range_input_producer

Produces the integers from 0 to limit-1 in a queue. (deprecated)

```
tf.compat.v1.train.range_input_producer(
    limit,
    num_epochs=None,
    shuffle=True,
    seed=None,
    capacity=32,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.range(limit).shuffle(limit).repeat(num_epochs)`. If `shuffle=False`, omit the `.shuffle(...)`.**Note:** if `num_epochs` is not `None`, this function creates local counter `epochs`. Use `local_variables_initializer()` to initialize local variables.

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.
- **`shared_name`**: (optional). If set, this queue will be shared under the given name across multiple sessions.
- **`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.

Eager Compatibility
Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.remove_checkpoint

Removes a checkpoint given by `checkpoint_prefix`. (deprecated)

```
tf.compat.v1.train.remove_checkpoint(
    checkpoint_prefix,
    checkpoint_format_version=tf.train.SaverDef.V2,
    meta_graph_suffix='meta'
)
```

Defined in `python/training/checkpoint_management.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use standard file APIs to delete files with this prefix.

Args:

- **`checkpoint_prefix`**: The prefix of a V1 or V2 checkpoint. Typically the result of `Saver.save()` or that of `tf.train.latest_checkpoint()`, regardless of sharded/non-sharded or V1/V2.

- **checkpoint_format_version**: SaverDef.CheckpointFormatVersion, defaults toSaverDef.V2.
- **meta_graph_suffix**: Suffix for MetaGraphDef file. Defaults to 'meta'.

tf.compat.v1.train.replica_device_setter

Return a device function to use when building a Graph for replicas.

```
tf.compat.v1.train.replica_device_setter(
    ps_tasks=0,
    ps_device='/job:ps',
    worker_device='/job:worker',
    merge_devices=True,
    cluster=None,
    ps_ops=None,
    ps_strategy=None
)
```

Defined in python/training/device_setter.py.

Device Functions are used in with tf.device(device_function): statement to automatically assign devices to Operation objects as they are constructed, Device constraints are added from the inner-most context first, working outwards. The merging behavior adds constraints to fields that are yet unset by a more inner context. Currently the fields are (job, task, cpu/gpu).

If cluster is None, and ps_tasks is 0, the returned function is a no-op. Otherwise, the value of ps_tasks is derived from cluster.

By default, only Variable ops are placed on ps tasks, and the placement strategy is round-robin over all ps tasks. A custom ps_strategy may be used to do more intelligent placement, such astf.contrib.training.GreedyLoadBalancingStrategy.

For example,

```
# To build a cluster with two ps jobs on hosts ps0 and ps1, and 3 worker
# jobs on hosts worker0, worker1 and worker2.
cluster_spec = {
    "ps": ["ps0:2222", "ps1:2222"],
    "worker": ["worker0:2222", "worker1:2222", "worker2:2222"]}
with
tf.device(tf.compat.v1.train.replica_device_setter(cluster=cluster_spec)):
  # Build your graph
  v1 = tf.Variable(...)  # assigned to /job:ps/task:0
  v2 = tf.Variable(...)  # assigned to /job:ps/task:1
  v3 = tf.Variable(...)  # assigned to /job:ps/task:0
# Run compute
```

Args:

- **ps_tasks**: Number of tasks in the ps job. Ignored if cluster is provided.
- **ps_device**: String. Device of the ps job. If empty no ps job is used. Defaults to ps.
- **worker_device**: String. Device of the worker job. If empty no worker job is used.
- **merge_devices**: Boolean. If True, merges or only sets a device if the device constraint is completely unset. merges device specification rather than overriding them.
- **cluster**: ClusterDef proto or ClusterSpec.
- **ps_ops**: List of strings representing Operation types that need to be placed on ps devices. If None, defaults to STANDARD_PS_OPS.

- **`ps_strategy`**: A callable invoked for every ps `Operation` (i.e. matched by `ps_ops`), that takes the `Operation` and returns the ps task index to use. If `None`, defaults to a round-robin strategy across all `ps` devices.

Returns:
A function to pass to `tf.device()`.

Raises:
TypeError if `cluster` is not a dictionary or `ClusterDef` protocol buffer, or if `ps_strategy` is provided but not a callable.

tf.compat.v1.train.RMSPropOptimizer

- Contents
- Class RMSPropOptimizer
- __init__
- Methods
  - apply_gradients

Class `RMSPropOptimizer`

Optimizer that implements the RMSProp algorithm.
Inherits From: `Optimizer`
Defined in `python/training/rmsprop.py`.
See the paper.

`__init__`

```
__init__(
    learning_rate,
    decay=0.9,
    momentum=0.0,
    epsilon=1e-10,
    use_locking=False,
    centered=False,
    name='RMSProp'
)
```

Construct a new RMSProp optimizer.
Note that in the dense implementation of this algorithm, variables and their corresponding accumulators (momentum, gradient moving average, square gradient moving average) will be updated even if the gradient is zero (i.e. accumulators will decay, momentum will be applied). The sparse implementation (used when the gradient is an `IndexedSlices` object, typically because of `tf.gather` or an embedding lookup in the forward pass) will not update variable slices or their accumulators unless those slices were used in the forward pass (nor is there an "eventual" correction to account for these omitted updates). This leads to more efficient updates for large embedding lookup tables (where most of the slices are not accessed in a particular graph execution), but differs from the published algorithm.

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.

- **`centered`**: If True, gradients are normalized by the estimated variance of the gradient; if False, by the uncentered second moment. Setting this to True may help with training, but is slightly more expensive in terms of computation and memory. Defaults to False.
- **`name`**: Optional name prefix for the operations created when applying gradients. Defaults to "RMSProp".

Eager Compatibility

When eager execution is enabled, `learning_rate`, `decay`, `momentum`, and `epsilon` can each be a callable that takes no arguments and returns the actual value to use. This can be useful for changing these values across different invocations of optimizer functions.

Methods

`apply_gradients`

```
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.
- **`RuntimeError`**: If you should use `_distributed_apply()` instead.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    grad_loss=None
)
```

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 or a callable taking no arguments which returns the value to minimize. When eager execution is enabled it must be a callable.
- **`var_list`**: Optional list or tuple of `tf.Variable` 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- **`TypeError`**: If `var_list` contains anything else than `Variable` objects.
- **`ValueError`**: If some arguments are invalid.
- **`RuntimeError`**: If called with eager execution enabled and `loss` is not callable.

Eager Compatibility

When eager execution is enabled, `gate_gradients`, `aggregation_method`, and `colocate_gradients_with_ops` are ignored.

get_name

```
get_name()
```

get_slot

```
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.

get_slot_names

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.

See `get_slot()`.

Returns:

A list of strings.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility

When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

A list of variables which encode the current state of `Optimizer`.
Includes slot variables and additional global variables created by the optimizer in the current default graph.

Returns:
A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.Saver

- **Contents**
- Class Saver
- __init__
- Properties
  - last_checkpoints

Class `Saver`

Saves and restores variables.
Defined in `python/training/saver.py`.
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, no checkpoints are deleted from the filesystem but only the last one is kept in the `checkpoint` file. 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.compat.v1.train.Saver(...variables...)
```

```
# Launch the graph and train, saving the model every 1,000 steps.
sess = tf.compat.v1.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()`.

`__init__`

```
__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,
    defer_build=False,
    allow_empty=False,
    write_version=tf.train.SaverDef.V2,
    pad_step_number=False,
    save_relative_paths=False,
    filename=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.compat.v1.train.Saver({'v1': v1, 'v2': v2})

# Or pass them as a list.
```

```
saver = tf.compat.v1.train.Saver([v1, v2])
# Passing a list is equivalent to passing a dict with the variable op names
# as keys:
saver = tf.compat.v1.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`/`SaveableObject`, or a dictionary mapping names to `SaveableObject`s. If `None`, defaults to the list of all saveable objects.
- **`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 `BulkSaverBuilder()`.
- **`defer_build`**: If `True`, defer adding the save and restore ops to the `build()` call. In that case `build()` should be called before finalizing the graph or using the saver.
- **`allow_empty`**: If `False` (default) raise an error if there are no variables in the graph. Otherwise, construct the saver anyway and make it a no-op.
- **`write_version`**: controls what format to use when saving checkpoints. It also affects certain filepath matching logic. The V2 format is the recommended choice: it is much more optimized than V1 in terms of memory required and latency incurred during restore. Regardless of this flag, the Saver is able to restore from both V2 and V1 checkpoints.
- **`pad_step_number`**: if True, pads the global step number in the checkpoint filepaths to some fixed width (8 by default). This is turned off by default.
- **`save_relative_paths`**: If `True`, will write relative paths to the checkpoint state file. This is needed if the user wants to copy the checkpoint directory and reload from the copied directory.
- **`filename`**: If known at graph construction time, filename used for variable loading/saving.

Raises:

- **`TypeError`**: If `var_list` is invalid.
- **`ValueError`**: If any of the keys or values in `var_list` are not unique.
- **`RuntimeError`**: If eager execution is enabled and`var_list` does not specify a list of varialbes to save.

Eager Compatibility

When eager execution is enabled, `var_list` must specify a `list` or `dict` of variables to save. Otherwise, a `RuntimeError` will be raised.
Although Saver works in some cases when executing eagerly, it is fragile. Please switch to `tf.train.Checkpoint` or `tf.keras.Model.save_weights`, which perform a more robust object-based saving. These APIs will load checkpoints written by `Saver`.

Properties

`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.

Methods

`as_saver_def`

```
as_saver_def()
```

Generates a `SaverDef` representation of this saver.

Returns:
A `SaverDef` proto.

`build`

```
build()
```

`export_meta_graph`

```
export_meta_graph(
    filename=None,
    collection_list=None,
    as_text=False,
    export_scope=None,
    clear_devices=False,
    clear_extraneous_savers=False,
    strip_default_attrs=False,
    save_debug_info=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.
- **`export_scope`**: Optional `string`. Name scope to remove.
- **`clear_devices`**: Whether or not to clear the device field for an `Operation` or `Tensor` during export.
- **`clear_extraneous_savers`**: Remove any Saver-related information from the graph (both Save/Restore ops and SaverDefs) that are not associated with this Saver.
- **`strip_default_attrs`**: Boolean. If `True`, default-valued attributes will be removed from the NodeDefs. For a detailed guide, see [Stripping Default-Valued Attributes].
- **`save_debug_info`**: If `True`, save the GraphDebugInfo to a separate file, which in the same directory of filename and with `_debug` added before the file extension.

Returns:
A `MetaGraphDef` proto.

from_proto

```
@staticmethod
from_proto(
    saver_def,
    import_scope=None
)
```

Returns a `Saver` object created from `saver_def`.

Args:

- **`saver_def`**: a `SaverDef` protocol buffer.
- **`import_scope`**: Optional `string`. Name scope to use.

Returns:

A `Saver` built from saver_def.

recover_last_checkpoints

```
recover_last_checkpoints(checkpoint_paths)
```

Recovers the internal saver state after a crash.

This method is useful for recovering the "self._last_checkpoints" state.

Globs for the checkpoints pointed to by `checkpoint_paths`. If the files exist, use their mtime as the checkpoint timestamp.

Args:

- **`checkpoint_paths`**: a list of checkpoint paths.

restore

```
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. None in eager mode.
- **`save_path`**: Path where parameters were previously saved.

Raises:

- **`ValueError`**: If save_path is None or not a valid checkpoint.

save

```
save(
    sess,
    save_path,
    global_step=None,
    latest_filename=None,
    meta_graph_suffix='meta',
```

```
    write_meta_graph=True,
    write_state=True,
    strip_default_attrs=False,
    save_debug_info=False
)
```

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 prefix of the newly created checkpoint files. This string can be passed directly to a call to `restore()`.

Args:

- **`sess`**: A Session to use to save the variables.
- **`save_path`**: String. Prefix of filenames created for the checkpoint.
- **`global_step`**: If provided the global step number is appended to `save_path` to create the checkpoint filenames. 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 checkpoints. 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'.
- **`write_meta_graph`**: `Boolean` indicating whether or not to write the meta graph file.
- **`write_state`**: `Boolean` indicating whether or not to write the `CheckpointStateProto`.
- **`strip_default_attrs`**: Boolean. If `True`, default-valued attributes will be removed from the NodeDefs. For a detailed guide, see [Stripping Default-Valued Attributes].
- **`save_debug_info`**: If `True`, save the GraphDebugInfo to a separate file, which in the same directory of save_path and with `_debug` added before the file extension. This is only enabled when `write_meta_graph` is `True`

Returns:

A string: path prefix used for the checkpoint files. If the saver is sharded, this string ends with: '-?????-of-nnnnn' where 'nnnnn' is the number of shards created. If the saver is empty, returns None.

Raises:

- **`TypeError`**: If `sess` is not a `Session`.
- **`ValueError`**: If `latest_filename` contains path components, or if it collides with `save_path`.
- **`RuntimeError`**: If save and restore ops weren't built.

set_last_checkpoints

```
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.

set_last_checkpoints_with_time

```
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.

`to_proto`

```
to_proto(export_scope=None)
```

Converts this `Saver` to a `SaverDef` protocol buffer.

Args:

- **`export_scope`**: Optional `string`. Name scope to remove.

Returns:

A `SaverDef` protocol buffer.

tf.compat.v1.train.SaverDef

Class `SaverDef`

Defined in `core/protobuf/saver.proto`.

Properties

`filename_tensor_name`

`string filename_tensor_name`

`keep_checkpoint_every_n_hours`

`float keep_checkpoint_every_n_hours`

`max_to_keep`

`int32 max_to_keep`

`restore_op_name`

`string restore_op_name`

`save_tensor_name`

`string save_tensor_name`

`sharded`

`bool sharded`

`version`

`CheckpointFormatVersion version`

Class Members

- `CheckpointFormatVersion`
- `LEGACY = 0`
- `V1 = 1`
- `V2 = 2`

tf.compat.v1.train.Scaffold

- **Contents**
- Class Scaffold
  - Used in the guide:
- __init__
- Properties

Class `Scaffold`

Structure to create or gather pieces commonly needed to train a model.
Defined in `python/training/monitored_session.py`.

Used in the guide:

- Training checkpoints

When you build a model for training you usually need ops to initialize variables, a `Saver` to checkpoint them, an op to collect summaries for the visualizer, and so on.
Various libraries built on top of the core TensorFlow library take care of creating some or all of these pieces and storing them in well known collections in the graph. The `Scaffold` class helps pick these pieces from the graph collections, creating and adding them to the collections if needed.
If you call the scaffold constructor without any arguments, it will pick pieces from the collections, creating default ones if needed when `scaffold.finalize()` is called. You can pass arguments to the constructor to provide your own pieces. Pieces that you pass to the constructor are not added to the graph collections.
The following pieces are directly accessible as attributes of the `Scaffold` object:

- `saver`: A `tf.compat.v1.train.Saver` object taking care of saving the variables. Picked from and stored into the `SAVERS` collection in the graph by default.
- `init_op`: An op to run to initialize the variables. Picked from and stored into the `INIT_OP`collection in the graph by default.
- `ready_op`: An op to verify that the variables are initialized. Picked from and stored into the `READY_OP` collection in the graph by default.
- `ready_for_local_init_op`: An op to verify that global state has been initialized and it is alright to run `local_init_op`. Picked from and stored into the `READY_FOR_LOCAL_INIT_OP` collection in the graph by default. This is needed when the initialization of local variables depends on the values of global variables.
- `local_init_op`: An op to initialize the local variables. Picked from and stored into the `LOCAL_INIT_OP` collection in the graph by default.
- `summary_op`: An op to run and merge the summaries in the graph. Picked from and stored into the `SUMMARY_OP` collection in the graph by default.

You can also pass the following additional pieces to the constructor:

- `init_feed_dict`: A session feed dictionary that should be used when running the init op.
- `init_fn`: A callable to run after the init op to perform additional initializations. The callable will be called as `init_fn(scaffold, session)`.

`__init__`

```
__init__(
    init_op=None,
    init_feed_dict=None,
    init_fn=None,
    ready_op=None,
    ready_for_local_init_op=None,
    local_init_op=None,
    summary_op=None,
    saver=None,
```

```
    copy_from_scaffold=None
)
```

Create a scaffold.

Args:

- **`init_op`**: Optional op for initializing variables.
- **`init_feed_dict`**: Optional session feed dictionary to use when running the init_op.
- **`init_fn`**: Optional function to use to initialize the model after running the init_op. Will be called as `init_fn(scaffold, session)`.
- **`ready_op`**: Optional op to verify that the variables are initialized. Must return an empty 1D string tensor when the variables are initialized, or a non-empty 1D string tensor listing the names of the non-initialized variables.
- **`ready_for_local_init_op`**: Optional op to verify that the global variables are initialized and `local_init_op` can be run. Must return an empty 1D string tensor when the global variables are initialized, or a non-empty 1D string tensor listing the names of the non-initialized global variables.
- **`local_init_op`**: Optional op to initialize local variables.
- **`summary_op`**: Optional op to gather all summaries. Must return a scalar string tensor containing a serialized `Summary` proto.
- **`saver`**: Optional `tf.compat.v1.train.Saver` object to use to save and restore variables. May also be a `tf.train.Checkpoint` object, in which case object-based checkpoints are saved. This will also load some object-based checkpoints saved from elsewhere, but that loading may be fragile since it uses fixed keys rather than performing a full graph-based match. For example if a variable has two paths from the `Checkpoint` object because two `Model` objects share the `Layer` object that owns it, removing one `Model` may change the keys and break checkpoint loading through this API, whereas a graph-based match would match the variable through the other `Model`.
- **`copy_from_scaffold`**: Optional scaffold object to copy fields from. Its fields will be overwritten by the provided fields in this function.

Properties

init_feed_dict

init_fn

init_op

local_init_op

ready_for_local_init_op

ready_op

saver

summary_op

Methods

default_local_init_op

```
@staticmethod
default_local_init_op()
```

Returns an op that groups the default local init ops.

This op is used during session initialization when a Scaffold is initialized without specifying the local_init_op arg. It includes `tf.compat.v1.local_variables_initializer`,`tf.compat.v1.tables_initializer`, and also initializes local session resources.

Returns:

The default Scaffold local init op.

finalize

```
finalize()
```

Creates operations if needed and finalizes the graph.

get_or_default

```
@staticmethod
get_or_default(
    arg_name,
    collection_key,
    default_constructor
)
```

Get from cache or create a default operation.

tf.compat.v1.train.sdca_fprint

Computes fingerprints of the input strings.

```
tf.compat.v1.train.sdca_fprint(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_sdca_ops.py`.

Args:

- **`input`**: A `Tensor` of type `string`. vector of strings to compute fingerprints on.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

tf.compat.v1.train.sdca_optimizer

Distributed version of Stochastic Dual Coordinate Ascent (SDCA) optimizer for

```
tf.compat.v1.train.sdca_optimizer(
    sparse_example_indices,
    sparse_feature_indices,
    sparse_feature_values,
    dense_features,
    example_weights,
    example_labels,
    sparse_indices,
    sparse_weights,
    dense_weights,
    example_state_data,
```

```
    loss_type,
    l1,
    l2,
    num_loss_partitions,
    num_inner_iterations,
    adaptative=True,
    name=None
)
```

Defined in generated file: `python/ops/gen_sdca_ops.py`.
linear models with L1 + L2 regularization. As global optimization objective is strongly-convex, the optimizer optimizes the dual objective at each step. The optimizer applies each update one example at a time. Examples are sampled uniformly, and the optimizer is learning rate free and enjoys linear convergence rate.
[Proximal Stochastic Dual Coordinate Ascent](#).
Shai Shalev-Shwartz, Tong Zhang. 2012

$$LossObjective=\sum f_i(wx_i)+(l2/2)*|w|^2+l1*|w|$$

[Adding vs. Averaging in Distributed Primal-Dual Optimization](#).
Chenxin Ma, Virginia Smith, Martin Jaggi, Michael I. Jordan, Peter Richtarik, Martin Takac. 2015
[Stochastic Dual Coordinate Ascent with Adaptive Probabilities](#).
Dominik Csiba, Zheng Qu, Peter Richtarik. 2015

Args:

- **`sparse_example_indices`**: A list of `Tensor` objects with type `int64`. a list of vectors which contain example indices.
- **`sparse_feature_indices`**: A list with the same length as `sparse_example_indices` of `Tensor` objects with type `int64`. a list of vectors which contain feature indices.
- **`sparse_feature_values`**: A list of `Tensor` objects with type `float32`. a list of vectors which contains feature value associated with each feature group.
- **`dense_features`**: A list of `Tensor` objects with type `float32`. a list of matrices which contains the dense feature values.
- **`example_weights`**: A `Tensor` of type `float32`. a vector which contains the weight associated with each example.
- **`example_labels`**: A `Tensor` of type `float32`. a vector which contains the label/target associated with each example.
- **`sparse_indices`**: A list with the same length as `sparse_example_indices` of `Tensor` objects with type `int64`. a list of vectors where each value is the indices which has corresponding weights in sparse_weights. This field maybe omitted for the dense approach.
- **`sparse_weights`**: A list with the same length as `sparse_example_indices` of `Tensor` objects with type `float32`. a list of vectors where each value is the weight associated with a sparse feature group.
- **`dense_weights`**: A list with the same length as `dense_features` of `Tensor` objects with type `float32`. a list of vectors where the values are the weights associated with a dense feature group.
- **`example_state_data`**: A `Tensor` of type `float32`. a list of vectors containing the example state data.
- **`loss_type`**: A `string` from: `"logistic_loss", "squared_loss", "hinge_loss", "smooth_hinge_loss", "poisson_loss"`. Type of the primal loss. Currently SdcaSolver supports logistic, squared and hinge losses.
- **`l1`**: A `float`. Symmetric l1 regularization strength.
- **`l2`**: A `float`. Symmetric l2 regularization strength.

- `num_loss_partitions`: An `int` that is `>= 1`. Number of partitions of the global loss function.
- `num_inner_iterations`: An `int` that is `>= 1`. Number of iterations per mini-batch.
- `adaptative`: An optional `bool`. Defaults to `True`. Whether to use Adaptive SDCA for the inner loop.
- `name`: A name for the operation (optional).

Returns:
A tuple of `Tensor` objects (out_example_state_data, out_delta_sparse_weights, out_delta_dense_weights).
- `out_example_state_data`: A `Tensor` of type `float32`.
- `out_delta_sparse_weights`: A list with the same length as `sparse_example_indices` of `Tensor` objects with type `float32`.
- `out_delta_dense_weights`: A list with the same length as `dense_features` of `Tensor`objects with type `float32`.

tf.compat.v1.train.sdca_shrink_l1

Applies L1 regularization shrink step on the parameters.

```
tf.compat.v1.train.sdca_shrink_l1(
    weights,
    l1,
    l2,
    name=None
)
```

Defined in generated file: `python/ops/gen_sdca_ops.py`.

Args:
- `weights`: A list of `Tensor` objects with type mutable `float32`. a list of vectors where each value is the weight associated with a feature group.
- `l1`: A `float`. Symmetric l1 regularization strength.
- `l2`: A `float`. Symmetric l2 regularization strength. Should be a positive float.
- `name`: A name for the operation (optional).

Returns:
The created Operation.

tf.compat.v1.train.SessionCreator

- **Contents**
- Class SessionCreator
- Methods
  - create_session

Class `SessionCreator`

A factory for tf.Session.
Defined in `python/training/monitored_session.py`.

Methods

`create_session`

```
create_session()
```

tf.compat.v1.train.SessionManager

- **Contents**
- Class SessionManager

- Usage:
- __init__
- Methods

Class `SessionManager`

Training helper that restores from checkpoint and creates session.

Defined in `python/training/session_manager.py`.

This class is a small wrapper that takes care of session creation and checkpoint recovery. It also provides functions that to facilitate coordination among multiple training threads or processes.

- Checkpointing trained variables as the training progresses.
- Initializing variables on startup, restoring them from the most recent checkpoint after a crash, or wait for checkpoints to become available.

Usage:

```
with tf.Graph().as_default():
   ...add operations to the graph...
  # Create a SessionManager that will checkpoint the model in '/tmp/mydir'.
  sm = SessionManager()
  sess = sm.prepare_session(master, init_op, saver, checkpoint_dir)
  # Use the session to train the graph.
  while True:
    sess.run(<my_train_op>)
```

`prepare_session()` initializes or restores a model. It requires `init_op` and `saver` as an argument.

A second process could wait for the model to be ready by doing the following:

```
with tf.Graph().as_default():
   ...add operations to the graph...
  # Create a SessionManager that will wait for the model to become ready.
  sm = SessionManager()
  sess = sm.wait_for_session(master)
  # Use the session to train the graph.
  while True:
    sess.run(<my_train_op>)
```

`wait_for_session()` waits for a model to be initialized by other processes.

`__init__`

```
__init__(
    local_init_op=None,
    ready_op=None,
    ready_for_local_init_op=None,
    graph=None,
    recovery_wait_secs=30,
    local_init_run_options=None
)
```

Creates a SessionManager.

The `local_init_op` is an `Operation` that is run always after a new session was created. If `None`, this step is skipped.

The `ready_op` is an `Operation` used to check if the model is ready. The model is considered ready if that operation returns an empty 1D string tensor. If the operation returns a non empty 1D string tensor, the elements are concatenated and used to indicate to the user why the model is not ready.
The `ready_for_local_init_op` is an `Operation` used to check if the model is ready to run local_init_op. The model is considered ready if that operation returns an empty 1D string tensor. If the operation returns a non empty 1D string tensor, the elements are concatenated and used to indicate to the user why the model is not ready.
If `ready_op` is `None`, the model is not checked for readiness.
`recovery_wait_secs` is the number of seconds between checks that the model is ready. It is used by processes to wait for a model to be initialized or restored. Defaults to 30 seconds.

Args:

- **`local_init_op`**: An `Operation` run immediately after session creation. Usually used to initialize tables and local variables.
- **`ready_op`**: An `Operation` to check if the model is initialized.
- **`ready_for_local_init_op`**: An `Operation` to check if the model is ready to run local_init_op.
- **`graph`**: The `Graph` that the model will use.
- **`recovery_wait_secs`**: Seconds between checks for the model to be ready.
- **`local_init_run_options`**: RunOptions to be passed to session.run when executing the local_init_op.

Raises:

- **`ValueError`**: If ready_for_local_init_op is not None but local_init_op is None

Methods

`prepare_session`

```
prepare_session(
    master,
    init_op=None,
    saver=None,
    checkpoint_dir=None,
    checkpoint_filename_with_path=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    config=None,
    init_feed_dict=None,
    init_fn=None
)
```

Creates a `Session`. Makes sure the model is ready to be used.
Creates a `Session` on 'master'. If a `saver` object is passed in, and `checkpoint_dir` points to a directory containing valid checkpoint files, then it will try to recover the model from checkpoint. If no checkpoint files are available, and `wait_for_checkpoint` is `True`, then the process would check every `recovery_wait_secs`, up to `max_wait_secs`, for recovery to succeed.
If the model cannot be recovered successfully then it is initialized by running the `init_op` and calling `init_fn` if they are provided. The `local_init_op` is also run after init_op and init_fn, regardless of whether the model was recovered successfully, but only if `ready_for_local_init_op` passes.
If the model is recovered from a checkpoint it is assumed that all global variables have been initialized, in particular neither `init_op` nor `init_fn` will be executed.
It is an error if the model cannot be recovered and no `init_op` or `init_fn` or `local_init_op` are passed.

Args:

- **`master`**: `String` representation of the TensorFlow master to use.
- **`init_op`**: Optional `Operation` used to initialize the model.
- **`saver`**: A `Saver` object used to restore a model.
- **`checkpoint_dir`**: Path to the checkpoint files. The latest checkpoint in the dir will be used to restore.
- **`checkpoint_filename_with_path`**: Full file name path to the checkpoint file.
- **`wait_for_checkpoint`**: Whether to wait for checkpoint to become available.
- **`max_wait_secs`**: Maximum time to wait for checkpoints to become available.
- **`config`**: Optional `ConfigProto` proto used to configure the session.
- **`init_feed_dict`**: Optional dictionary that maps `Tensor` objects to feed values. This feed dictionary is passed to the session `run()` call when running the init op.
- **`init_fn`**: Optional callable used to initialize the model. Called after the optional `init_op` is called. The callable must accept one argument, the session being initialized.

Returns:

A `Session` object that can be used to drive the model.

Raises:

- **`RuntimeError`**: If the model cannot be initialized or recovered.
- **`ValueError`**: If both checkpoint_dir and checkpoint_filename_with_path are set.

`recover_session`

```
recover_session(
    master,
    saver=None,
    checkpoint_dir=None,
    checkpoint_filename_with_path=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    config=None
)
```

Creates a `Session`, recovering if possible.

Creates a new session on 'master'. If the session is not initialized and can be recovered from a checkpoint, recover it.

Args:

- **`master`**: `String` representation of the TensorFlow master to use.
- **`saver`**: A `Saver` object used to restore a model.
- **`checkpoint_dir`**: Path to the checkpoint files. The latest checkpoint in the dir will be used to restore.
- **`checkpoint_filename_with_path`**: Full file name path to the checkpoint file.
- **`wait_for_checkpoint`**: Whether to wait for checkpoint to become available.
- **`max_wait_secs`**: Maximum time to wait for checkpoints to become available.
- **`config`**: Optional `ConfigProto` proto used to configure the session.

Returns:

A pair (sess, initialized) where 'initialized' is `True` if the session could be recovered and initialized, `False` otherwise.

Raises:

- **`ValueError`**: If both checkpoint_dir and checkpoint_filename_with_path are set.

wait_for_session

```
wait_for_session(
    master,
    config=None,
    max_wait_secs=float('Inf')
)
```

Creates a new `Session` and waits for model to be ready.
Creates a new `Session` on 'master'. Waits for the model to be initialized or recovered from a checkpoint. It's expected that another thread or process will make the model ready, and that this is intended to be used by threads/processes that participate in a distributed training configuration where a different thread/process is responsible for initializing or recovering the model being trained.
NB: The amount of time this method waits for the session is bounded by max_wait_secs. By default, this function will wait indefinitely.

Args:

- **`master`**: `String` representation of the TensorFlow master to use.
- **`config`**: Optional ConfigProto proto used to configure the session.
- **`max_wait_secs`**: Maximum time to wait for the session to become available.

Returns:

A `Session`. May be None if the operation exceeds the timeout specified by config.operation_timeout_in_ms.

Raises:

- **`tf.DeadlineExceededError`**: if the session is not available after max_wait_secs.

tf.compat.v1.train.shuffle_batch

Creates batches by randomly shuffling tensors. (deprecated)

```
tf.compat.v1.train.shuffle_batch(
    tensors,
    batch_size,
    capacity,
    min_after_dequeue,
    num_threads=1,
    seed=None,
    enqueue_many=False,
    shapes=None,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by **`tf.data`**.
Use **`tf.data.Dataset.shuffle(min_after_dequeue).batch(batch_size)`**.
This function adds the following to the current `Graph`:

- A shuffling queue into which tensors from `tensors` are enqueued.
- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensors`.

If `enqueue_many` is `False`, `tensors` 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`, `tensors` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors` 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.compat.v1.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 `tensors` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.
If `allow_smaller_final_batch` is `True`, a smaller batch value than `batch_size` is returned when the queue is closed and there are not enough elements to fill the batch, otherwise the pending elements are discarded. In addition, all output tensors' static shapes, as accessed via the `shape`property will have a first `Dimension` value of `None`, and operations that depend on fixed batch_size would fail.

Args:

- **`tensors`**: The list or dictionary 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`.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (Optional) If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the types as `tensors`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

Eager Compatibility
Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.shuffle_batch_join

Create batches by randomly shuffling tensors. (deprecated)

```
tf.compat.v1.train.shuffle_batch_join(
    tensors_list,
    batch_size,
    capacity,
    min_after_dequeue,
    seed=None,
    enqueue_many=False,
    shapes=None,
    allow_smaller_final_batch=False,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.interleave(...).shuffle(min_after_dequeue).batch(batch_size)`.

The `tensors_list` argument is a list of tuples of tensors, or a list of dictionaries of tensors. Each element in the list is treated similarly to the `tensors` argument of `tf.compat.v1.train.shuffle_batch()`.

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 `tensors_list` are enqueued.
- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensors_list`.

`len(tensors_list)` threads will be started, with thread `i` enqueuing the tensors from `tensors_list[i]`. `tensors_list[i1][j]` must match `tensors_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensors_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`, `tensors_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors_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.

If `allow_smaller_final_batch` is `True`, a smaller batch value than `batch_size` is returned when the queue is closed and there are not enough elements to fill the batch, otherwise the pending elements are discarded. In addition, all output tensors' static shapes, as accessed via the `shape`property will have a first `Dimension` value of `None`, and operations that depend on fixed batch_size would fail.

Args:

- **`tensors_list`**: A list of tuples or dictionaries 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 `tensors_list[i]`.
- **`allow_smaller_final_batch`**: (Optional) Boolean. If `True`, allow the final batch to be smaller if there are insufficient items left in the queue.
- **`shared_name`**: (optional). If set, this queue will be shared under the given name across multiple sessions.
- **`name`**: (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- **`ValueError`**: If the `shapes` are not specified, and cannot be inferred from the elements of `tensors_list`.

Eager Compatibility

Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.SingularMonitoredSession

- **Contents**
- Class SingularMonitoredSession
- __init__
- Child Classes
- Properties

Class `SingularMonitoredSession`

Session-like object that handles initialization, restoring, and hooks.

Defined in `python/training/monitored_session.py`.

Please note that this utility is not recommended for distributed settings. For distributed settings, please use `tf.compat.v1.train.MonitoredSession`. The differences between `MonitoredSession`and `SingularMonitoredSession` are:

- `MonitoredSession` handles `AbortedError` and `UnavailableError` for distributed settings, but `SingularMonitoredSession` does not.
- `MonitoredSession` can be created in `chief` or `worker` modes. `SingularMonitoredSession`is always created as `chief`.
- You can access the raw `tf.compat.v1.Session` object used by `SingularMonitoredSession`, whereas in MonitoredSession the raw session is private. This can be used:
- To `run` without hooks.
- To save and restore.
- All other functionality is identical.

Example usage:

```
saver_hook = CheckpointSaverHook(...)
summary_hook = SummarySaverHook(...)
```

```
with SingularMonitoredSession(hooks=[saver_hook, summary_hook]) as sess:
  while not sess.should_stop():
    sess.run(train_op)
```

Initialization: At creation time the hooked session does following things in given order:

- calls `hook.begin()` for each given hook
- finalizes the graph via `scaffold.finalize()`
- create session
- initializes the model via initialization ops provided by `Scaffold`
- restores variables if a checkpoint exists
- launches queue runners

Run: When `run()` is called, the hooked session does following things:

- calls `hook.before_run()`
- calls TensorFlow `session.run()` with merged fetches and feed_dict
- calls `hook.after_run()`
- returns result of `session.run()` asked by user

Exit: At the `close()`, the hooked session does following things in order:

- calls `hook.end()`
- closes the queue runners and the session
- suppresses `OutOfRange` error which indicates that all inputs have been processed if the `SingularMonitoredSession` is used as a context.

`__init__`

```
__init__(
    hooks=None,
    scaffold=None,
    master='',
    config=None,
    checkpoint_dir=None,
    stop_grace_period_secs=120,
    checkpoint_filename_with_path=None
)
```

Creates a SingularMonitoredSession.

Args:

- **`hooks`**: An iterable of `SessionRunHook' objects.
- **`scaffold`**: A `Scaffold` used for gathering or building supportive ops. If not specified a default one is created. It's used to finalize the graph.
- **`master`**: `String` representation of the TensorFlow master to use.
- **`config`**: `ConfigProto` proto used to configure the session.
- **`checkpoint_dir`**: A string. Optional path to a directory where to restore variables.
- **`stop_grace_period_secs`**: Number of seconds given to threads to stop after `close()` has been called.
- **`checkpoint_filename_with_path`**: A string. Optional path to a checkpoint file from which to restore variables.

Child Classes

`class StepContext`

Properties

`graph`

The graph that was launched in this session.

Methods

`__enter__`

```
__enter__()
```

`__exit__`

```
__exit__(
    exception_type,
    exception_value,
    traceback
)
```

`close`

```
close()
```

`raw_session`

```
raw_session()
```

Returns underlying `TensorFlow.Session` object.

`run`

```
run(
    fetches,
    feed_dict=None,
    options=None,
    run_metadata=None
)
```

Run ops in the monitored session.
This method is completely compatible with the `tf.Session.run()` method.

Args:

- **`fetches`**: Same as `tf.Session.run()`.
- **`feed_dict`**: Same as `tf.Session.run()`.
- **`options`**: Same as `tf.Session.run()`.
- **`run_metadata`**: Same as `tf.Session.run()`.

Returns:
Same as `tf.Session.run()`.

`run_step_fn`

```
run_step_fn(step_fn)
```

Run ops using a step function.

Args:

- **`step_fn`**: A function or a method with a single argument of type `StepContext`. The function may use methods of the argument to perform computations with access to a raw session. The returned value of the `step_fn` will be returned from `run_step_fn`, unless a stop is requested. In that case, the next `should_stop` call will return True. Example usage: ```python with tf.Graph().as_default(): c = tf.compat.v1.placeholder(dtypes.float32) v = tf.add(c, 4.0) w = tf.add(c, 0.5) def step_fn(step_context): a = step_context.session.run(fetches=v, feed_dict={c: 0.5}) if a <= 4.5: step_context.request_stop() return step_context.run_with_hooks(fetches=w, feed_dict={c: 0.1}) with tf.MonitoredSession() as session: while not session.should_stop(): a = session.run_step_fn(step_fn)

```
    ```  Hooks interact with the `run_with_hooks()` call inside the
    `step_fn` as they do with a `MonitoredSession.run` call.

```

Returns:

Returns the returned value of `step_fn`.

Raises:

- **`StopIteration`**: if `step_fn` has called `request_stop()`. It may be caught by `with tf.MonitoredSession()` to close the session.
- **`ValueError`**: if `step_fn` doesn't have a single argument called `step_context`. It may also optionally have `self` for cases when it belongs to an object.

should_stop

```
should_stop()
```

tf.compat.v1.train.slice_input_producer

Produces a slice of each `Tensor` in `tensor_list`. (deprecated)

```
tf.compat.v1.train.slice_input_producer(
    tensor_list,
    num_epochs=None,
    shuffle=True,
    seed=None,
    capacity=32,
    shared_name=None,
    name=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.from_tensor_slices(tuple(tensor_list)).shuffle(tf.shape(input_tensor, out_type=tf.int64)[0]).repeat(num_epochs)`. If `shuffle=False`, omit the `.shuffle(...)`.

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.
- `shared_name`: (optional). If set, this queue will be shared under the given name across multiple sessions.
- `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`.

Eager Compatibility
Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.start_queue_runners

- **Contents**
- Aliases:

Starts all queue runners collected in the graph. (deprecated)

Aliases:
- `tf.compat.v1.train.queue_runner.start_queue_runners`
- `tf.compat.v1.train.start_queue_runners`

```
tf.compat.v1.train.start_queue_runners(
    sess=None,
    coord=None,
    daemon=True,
    start=True,
    collection=tf.GraphKeys.QUEUE_RUNNERS
)
```

Defined in `python/training/queue_runner_impl.py`.
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: To construct input pipelines, use the `tf.data` module.
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`.

Raises:
- `ValueError`: if `sess` is None and there isn't any default session.
- `TypeError`: if `sess` is not a `tf.compat.v1.Session` object.

Returns:
A list of threads.

Raises:

- `RuntimeError`: If called with eager execution enabled.
- `ValueError`: If called without a default `tf.compat.v1.Session` registered.

Eager Compatibility

Not compatible with eager execution. To ingest data under eager execution, use the `tf.data` API instead.

tf.compat.v1.train.string_input_producer

Output strings (e.g. filenames) to a queue for an input pipeline. (deprecated)

```
tf.compat.v1.train.string_input_producer(
    string_tensor,
    num_epochs=None,
    shuffle=True,
    seed=None,
    capacity=32,
    shared_name=None,
    name=None,
    cancel_op=None
)
```

Defined in `python/training/input.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Queue-based input pipelines have been replaced by `tf.data`. Use `tf.data.Dataset.from_tensor_slices(string_tensor).shuffle(tf.shape(input_tensor, out_type=tf.int64)[0]).repeat(num_epochs)`. If `shuffle=False`, omit the `.shuffle(...)`.**Note:** if `num_epochs` is not `None`, this function creates local counter `epochs`. Use `local_variables_initializer()` to initialize local variables.

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.
- `shared_name`: (optional). If set, this queue will be shared under the given name across multiple sessions. All sessions open to the device which has this queue will be able to access it via the shared_name. Using this in a distributed setting means each name will only be seen by one of the sessions which has access to this operation.
- `name`: A name for the operations (optional).
- `cancel_op`: Cancel op for the queue (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.

Eager Compatibility
Input pipelines based on Queues are not supported when eager execution is enabled. Please use the `tf.data` API to ingest data under eager execution.

tf.compat.v1.train.summary_iterator

An iterator for reading `Event` protocol buffers from an event file.

```
tf.compat.v1.train.summary_iterator(path)
```

Defined in `python/summary/summary_iterator.py`.
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.compat.v1.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.compat.v1.summary.scalar('loss', loss_tensor)`.
for e in tf.compat.v1.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.

tf.compat.v1.train.Supervisor

- **Contents**
- Class Supervisor
- __init__
- Properties
  - coord

Class `Supervisor`

A training helper that checkpoints models and computes summaries.
Defined in `python/training/supervisor.py`.
This class is deprecated. Please use `tf.compat.v1.train.MonitoredTrainingSession` instead.
The Supervisor is a small wrapper around a `Coordinator`, a `Saver`, and a `SessionManager` that takes care of common needs of TensorFlow training programs.

Use for a single program

```
with tf.Graph().as_default():
  ...add operations to the graph...
  # Create a Supervisor that will checkpoint the model in '/tmp/mydir'.
  sv = Supervisor(logdir='/tmp/mydir')
```

```
  # Get a TensorFlow session managed by the supervisor.
  with sv.managed_session(FLAGS.master) as sess:
    # Use the session to train the graph.
    while not sv.should_stop():
      sess.run(<my_train_op>)
```

Within the `with sv.managed_session()` block all variables in the graph have been initialized. In addition, a few services have been started to checkpoint the model and add summaries to the event log.
If the program crashes and is restarted, the managed session automatically reinitialize variables from the most recent checkpoint.
The supervisor is notified of any exception raised by one of the services. After an exception is raised, `should_stop()` returns `True`. In that case the training loop should also stop. This is why the training loop has to check for `sv.should_stop()`.
Exceptions that indicate that the training inputs have been exhausted, `tf.errors.OutOfRangeError`, also cause `sv.should_stop()` to return `True` but are not re-raised from the `with` block: they indicate a normal termination.

Use for multiple replicas
To train with replicas you deploy the same program in a `Cluster`. One of the tasks must be identified as the *chief*: the task that handles initialization, checkpoints, summaries, and recovery. The other tasks depend on the *chief* for these services.
The only change you have to do to the single program code is to indicate if the program is running as the *chief*.

```
# Choose a task as the chief. This could be based on server_def.task_index,
# or job_def.name, or job_def.tasks. It's entirely up to the end user.
# But there can be only one *chief*.
is_chief = (server_def.task_index == 0)
server = tf.distribute.Server(server_def)

with tf.Graph().as_default():
  ...add operations to the graph...
  # Create a Supervisor that uses log directory on a shared file system.
  # Indicate if you are the 'chief'
  sv = Supervisor(logdir='/shared_directory/...', is_chief=is_chief)
  # Get a Session in a TensorFlow server on the cluster.
  with sv.managed_session(server.target) as sess:
    # Use the session to train the graph.
    while not sv.should_stop():
      sess.run(<my_train_op>)
```

In the *chief* task, the `Supervisor` works exactly as in the first example above. In the other tasks `sv.managed_session()` waits for the Model to have been initialized before returning a session to the training code. The non-chief tasks depend on the chief task for initializing the model.
If one of the tasks crashes and restarts, `managed_session()` checks if the Model is initialized. If yes, it just creates a session and returns it to the training code that proceeds normally. If the model needs to be initialized, the chief task takes care of reinitializing it; the other tasks just wait for the model to have been initialized.
NOTE: This modified program still works fine as a single program. The single program marks itself as the chief.

What `master` string to use
Whether you are running on your machine or in the cluster you can use the following values for the --master flag:

- Specifying `''` requests an in-process session that does not use RPC.
- Specifying `'local'` requests a session that uses the RPC-based "Master interface" to run TensorFlow programs. See `tf.train.Server.create_local_server` for details.
- Specifying `'grpc://hostname:port'` requests a session that uses the RPC interface to a specific host, and also allows the in-process master to access remote tensorflow workers. Often, it is appropriate to pass `server.target` (for some `tf.distribute.Server` named `server).

Advanced use

Launching additional services
`managed_session()` launches the Checkpoint and Summary services (threads). If you need more services to run you can simply launch them in the block controlled by `managed_session()`.
Example: Start a thread to print losses. We want this thread to run every 60 seconds, so we launch it with `sv.loop()`.

```
...
sv = Supervisor(logdir='/tmp/mydir')
with sv.managed_session(FLAGS.master) as sess:
  sv.loop(60, print_loss, (sess, ))
  while not sv.should_stop():
    sess.run(my_train_op)
```

Launching fewer services
`managed_session()` launches the "summary" and "checkpoint" threads which use either the optionally `summary_op` and `saver` passed to the constructor, or default ones created automatically by the supervisor. If you want to run your own summary and checkpointing logic, disable these services by passing `None` to the `summary_op` and `saver` parameters.
Example: Create summaries manually every 100 steps in the chief.

```
# Create a Supervisor with no automatic summaries.
sv = Supervisor(logdir='/tmp/mydir', is_chief=is_chief, summary_op=None)
# As summary_op was None, managed_session() does not start the
# summary thread.
with sv.managed_session(FLAGS.master) as sess:
  for step in xrange(1000000):
    if sv.should_stop():
      break
    if is_chief and step % 100 == 0:
      # Create the summary every 100 chief steps.
      sv.summary_computed(sess, sess.run(my_summary_op))
    else:
      # Train normally
      sess.run(my_train_op)
```

Custom model initialization
`managed_session()` only supports initializing the model by running an `init_op` or restoring from the latest checkpoint. If you have special initialization needs, see how to specify a `local_init_op`when creating the supervisor. You can also use the `SessionManager` directly to create a session and check if it could be initialized automatically.

__init__

```
__init__(
    graph=None,
    ready_op=USE_DEFAULT,
    ready_for_local_init_op=USE_DEFAULT,
    is_chief=True,
    init_op=USE_DEFAULT,
    init_feed_dict=None,
    local_init_op=USE_DEFAULT,
    logdir=None,
    summary_op=USE_DEFAULT,
    saver=USE_DEFAULT,
    global_step=USE_DEFAULT,
    save_summaries_secs=120,
    save_model_secs=600,
    recovery_wait_secs=30,
    stop_grace_secs=120,
    checkpoint_basename='model.ckpt',
    session_manager=None,
    summary_writer=USE_DEFAULT,
    init_fn=None,
    local_init_run_options=None
)
```

Create a `Supervisor`. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Please switch to tf.train.MonitoredTrainingSession

Args:

- **`graph`**: A `Graph`. The graph that the model will use. Defaults to the default `Graph`. The supervisor may add operations to the graph before creating a session, but the graph should not be modified by the caller after passing it to the supervisor.
- **`ready_op`**: 1-D string `Tensor`. This tensor is evaluated by supervisors in`prepare_or_wait_for_session()` to check if the model is ready to use. The model is considered ready if it returns an empty array. Defaults to the tensor returned from `tf.compat.v1.report_uninitialized_variables()` If `None`, the model is not checked for readiness.
- **`ready_for_local_init_op`**: 1-D string `Tensor`. This tensor is evaluated by supervisors in `prepare_or_wait_for_session()` to check if the model is ready to run the local_init_op. The model is considered ready if it returns an empty array. Defaults to `None`. If `None`, the model is not checked for readiness before running local_init_op.
- **`is_chief`**: If True, create a chief supervisor in charge of initializing and restoring the model. If False, create a supervisor that relies on a chief supervisor for inits and restore.
- **`init_op`**: `Operation`. Used by chief supervisors to initialize the model when it can not be recovered. Defaults to an `Operation` that initializes all global variables. If `None`, no initialization is done automatically unless you pass a value for `init_fn`, see below.
- **`init_feed_dict`**: A dictionary that maps `Tensor` objects to feed values. This feed dictionary will be used when `init_op` is evaluated.

- **`local_init_op`**: `Operation`. Used by all supervisors to run initializations that should run for every new supervisor instance. By default these are table initializers and initializers for local variables. If `None`, no further per supervisor-instance initialization is done automatically.
- **`logdir`**: A string. Optional path to a directory where to checkpoint the model and log events for the visualizer. Used by chief supervisors. The directory will be created if it does not exist.
- **`summary_op`**: An `Operation` that returns a Summary for the event logs. Used by chief supervisors if a `logdir` was specified. Defaults to the operation returned from summary.merge_all(). If `None`, summaries are not computed automatically.
- **`saver`**: A Saver object. Used by chief supervisors if a `logdir` was specified. Defaults to the saved returned by Saver(). If `None`, the model is not saved automatically.
- **`global_step`**: An integer Tensor of size 1 that counts steps. The value from 'global_step' is used in summaries and checkpoint filenames. Default to the op named 'global_step' in the graph if it exists, is of rank 1, size 1, and of type tf.int32 or tf.int64. If `None` the global step is not recorded in summaries and checkpoint files. Used by chief supervisors if a `logdir` was specified.
- **`save_summaries_secs`**: Number of seconds between the computation of summaries for the event log. Defaults to 120 seconds. Pass 0 to disable summaries.
- **`save_model_secs`**: Number of seconds between the creation of model checkpoints. Defaults to 600 seconds. Pass 0 to disable checkpoints.
- **`recovery_wait_secs`**: Number of seconds between checks that the model is ready. Used by supervisors when waiting for a chief supervisor to initialize or restore the model. Defaults to 30 seconds.
- **`stop_grace_secs`**: Grace period, in seconds, given to running threads to stop when `stop()` is called. Defaults to 120 seconds.
- **`checkpoint_basename`**: The basename for checkpoint saving.
- **`session_manager`**: `SessionManager`, which manages Session creation and recovery. If it is `None`, a default `SessionManager` will be created with the set of arguments passed in for backwards compatibility.
- **`summary_writer`**: `SummaryWriter` to use or `USE_DEFAULT`. Can be `None` to indicate that no summaries should be written.
- **`init_fn`**: Optional callable used to initialize the model. Called after the optional `init_op` is called. The callable must accept one argument, the session being initialized.
- **`local_init_run_options`**: RunOptions to be passed as the SessionManager local_init_run_options parameter.

Returns:
A `Supervisor`.

Raises:
- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility
`Supervisor`s are not supported when eager execution is enabled.

Properties

`coord`

Return the Coordinator used by the Supervisor.
The Coordinator can be useful if you want to run multiple threads during your training.

Returns:
A Coordinator object.

`global_step`

Return the global_step Tensor used by the supervisor.

Returns:
An integer Tensor for the global_step.

`init_feed_dict`
Return the feed dictionary used when evaluating the `init_op`.

Returns:
A feed dictionary or `None`.

`init_op`
Return the Init Op used by the supervisor.

Returns:
An Op or `None`.

`is_chief`
Return True if this is a chief supervisor.

Returns:
A bool.

`ready_for_local_init_op`

`ready_op`
Return the Ready Op used by the supervisor.

Returns:
An Op or `None`.

`save_model_secs`
Return the delay between checkpoints.

Returns:
A timestamp.

`save_path`
Return the save path used by the supervisor.

Returns:
A string.

`save_summaries_secs`
Return the delay between summary computations.

Returns:
A timestamp.

`saver`
Return the Saver used by the supervisor.

Returns:
A Saver object.

`session_manager`
Return the SessionManager used by the Supervisor.

Returns:
A SessionManager object.

`summary_op`
Return the Summary Tensor used by the chief supervisor.

Returns:
A string Tensor for the summary or `None`.

`summary_writer`

Return the SummaryWriter used by the chief supervisor.

Returns:
A SummaryWriter.

Methods

`Loop`

```
Loop(
    timer_interval_secs,
    target,
    args=None,
    kwargs=None
)
```

Start a LooperThread that calls a function periodically.
If `timer_interval_secs` is None the thread calls `target(*args, **kwargs)` repeatedly. Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.
The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- **`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.
- **`kwargs`**: Optional keyword arguments to pass to `target` when calling it.

Returns:
The started thread.

`PrepareSession`

```
PrepareSession(
    master='',
    config=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    start_standard_services=True
)
```

Make sure the model is ready to be used.
Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- **`master`**: name of the TensorFlow master to use. See the `tf.compat.v1.Session` constructor for how this is interpreted.
- **`config`**: Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.

- **`wait_for_checkpoint`**: Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- **`max_wait_secs`**: Maximum time to wait for the session to become available.
- **`start_standard_services`**: Whether to start the standard services and the queue runners.

Returns:
A Session object that can be used to drive the model.

RequestStop

```
RequestStop(ex=None)
```

Request that the coordinator stop the threads.
See `Coordinator.request_stop()`.

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()`.

ShouldStop

```
ShouldStop()
```

Check if the coordinator was told to stop.
See `Coordinator.should_stop()`.

Returns:
True if the coordinator was told to stop, False otherwise.

StartQueueRunners

```
StartQueueRunners(
    sess,
    queue_runners=None
)
```

Start threads for `QueueRunners`.
Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- **`sess`**: A `Session`.
- **`queue_runners`**: A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:
The list of threads started for the `QueueRunners`.

Raises:

- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility
Queues are not compatible with eager execution. To ingest data when eager execution is enabled, use the `tf.data` API.

StartStandardServices

```
StartStandardServices(sess)
```

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every save_summaries_secs.
- A Checkpoint thread saving the model every save_model_secs.
- A StepCounter thread measure step time.

Args:

- **`sess`**: A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: sv.coord.Join()

Raises:

- **`RuntimeError`**: If called with a non-chief Supervisor.
- **`ValueError`**: If not `logdir` was passed to the constructor as the services need a log directory.

Stop

```
Stop(
    threads=None,
    close_summary_writer=True,
    ignore_live_threads=False
)
```

Stop the services and the coordinator.
This does not close the session.

Args:

- **`threads`**: Optional list of threads to join with the coordinator. If `None`, defaults to the threads running the standard services, the threads started for `QueueRunners`, and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
- **`close_summary_writer`**: Whether to close the `summary_writer`. Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.
- **`ignore_live_threads`**: If `True` ignores threads that remain running after a grace period when joining threads via the coordinator, instead of raising a RuntimeError.

StopOnException

```
StopOnException()
```

Context handler to stop the supervisor when an exception is raised.
See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

SummaryComputed

```
SummaryComputed(
    sess,
    summary,
    global_step=None
)
```

Indicate that a summary was computed.

Args:

- **`sess`**: A `Session` object.

- **`summary`**: A Summary proto, or a string holding a serialized summary proto.
- **`global_step`**: Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- **`TypeError`**: if 'summary' is not a Summary proto or a string.
- **`RuntimeError`**: if the Supervisor was created without a `logdir`.

WaitForStop

```
WaitForStop()
```

Block waiting for the coordinator to stop.

loop

```
loop(
    timer_interval_secs,
    target,
    args=None,
    kwargs=None
)
```

Start a LooperThread that calls a function periodically.
If `timer_interval_secs` is None the thread calls `target(*args, **kwargs)` repeatedly. Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.
The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- **`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.
- **`kwargs`**: Optional keyword arguments to pass to `target` when calling it.

Returns:
The started thread.

managed_session

```
managed_session(
    *args,
    **kwds
)
```

Returns a context manager for a managed session.
This context manager creates and automatically recovers a session. It optionally starts the standard services that handle checkpoints and summaries. It monitors exceptions raised from the `with` block or from the services and stops the supervisor as needed.
The context manager is typically used as follows:

```
def train():
  sv = tf.compat.v1.train.Supervisor(...)
  with sv.managed_session(<master>) as sess:
    for step in xrange(..):
      if sv.should_stop():
```

```
        break
      sess.run(<my training op>)
      ...do other things needed at each training step...
```

An exception raised from the `with` block or one of the service threads is raised again when the block exits. This is done after stopping all threads and closing the session. For example, an `AbortedError`exception, raised in case of preemption of one of the workers in a distributed model, is raised again when the block exits.
If you want to retry the training loop in case of preemption you can do it as follows:

```
def main(...):
  while True
    try:
      train()
    except tf.errors.Aborted:
      pass
```

As a special case, exceptions used for control flow, such as `OutOfRangeError` which reports that input queues are exhausted, are not raised again from the `with` block: they indicate a clean termination of the training loop and are considered normal termination.

Args:

- **`master`**: name of the TensorFlow master to use. See the `tf.compat.v1.Session` constructor for how this is interpreted.
- **`config`**: Optional `ConfigProto` proto used to configure the session. Passed as-is to create the session.
- **`start_standard_services`**: Whether to start the standard services, such as checkpoint, summary and step counter.
- **`close_summary_writer`**: Whether to close the summary writer when closing the session. Defaults to True.

Returns:

A context manager that yields a `Session` restored from the latest checkpoint or initialized from scratch if not checkpoint exists. The session is closed when the `with` block exits.

prepare_or_wait_for_session

```
prepare_or_wait_for_session(
    master='',
    config=None,
    wait_for_checkpoint=False,
    max_wait_secs=7200,
    start_standard_services=True
)
```

Make sure the model is ready to be used.
Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- **`master`**: name of the TensorFlow master to use. See the `tf.compat.v1.Session` constructor for how this is interpreted.
- **`config`**: Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.

- **`wait_for_checkpoint`**: Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- **`max_wait_secs`**: Maximum time to wait for the session to become available.
- **`start_standard_services`**: Whether to start the standard services and the queue runners.

Returns:
A Session object that can be used to drive the model.

`request_stop`

```
request_stop(ex=None)
```

Request that the coordinator stop the threads.
See `Coordinator.request_stop()`.

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()`.

`should_stop`

```
should_stop()
```

Check if the coordinator was told to stop.
See `Coordinator.should_stop()`.

Returns:
True if the coordinator was told to stop, False otherwise.

`start_queue_runners`

```
start_queue_runners(
    sess,
    queue_runners=None
)
```

Start threads for `QueueRunners`.
Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- **`sess`**: A `Session`.
- **`queue_runners`**: A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:
The list of threads started for the `QueueRunners`.

Raises:

- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility
Queues are not compatible with eager execution. To ingest data when eager execution is enabled, use the `tf.data` API.

`start_standard_services`

```
start_standard_services(sess)
```

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every save_summaries_secs.
- A Checkpoint thread saving the model every save_model_secs.
- A StepCounter thread measure step time.

Args:

- **`sess`**: A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: sv.coord.Join()

Raises:

- **`RuntimeError`**: If called with a non-chief Supervisor.
- **`ValueError`**: If not `logdir` was passed to the constructor as the services need a log directory.

`stop`

```
stop(
    threads=None,
    close_summary_writer=True,
    ignore_live_threads=False
)
```

Stop the services and the coordinator.

This does not close the session.

Args:

- **`threads`**: Optional list of threads to join with the coordinator. If `None`, defaults to the threads running the standard services, the threads started for `QueueRunners`, and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
- **`close_summary_writer`**: Whether to close the `summary_writer`. Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.
- **`ignore_live_threads`**: If `True` ignores threads that remain running after a grace period when joining threads via the coordinator, instead of raising a RuntimeError.

`stop_on_exception`

```
stop_on_exception()
```

Context handler to stop the supervisor when an exception is raised.

See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

`summary_computed`

```
summary_computed(
    sess,
    summary,
    global_step=None
)
```

Indicate that a summary was computed.

Args:

- **`sess`**: A `Session` object.

- `summary`: A Summary proto, or a string holding a serialized summary proto.
- `global_step`: Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- `TypeError`: if 'summary' is not a Summary proto or a string.
- `RuntimeError`: if the Supervisor was created without a `logdir`.

`wait_for_stop`

```
wait_for_stop()
```

Block waiting for the coordinator to stop.

Class Members

- `USE_DEFAULT = 0`

tf.compat.v1.train.SyncReplicasOptimizer

- **Contents**
- Class SyncReplicasOptimizer
  - Usage
- __init__
- Methods

Class `SyncReplicasOptimizer`

Class to synchronize, aggregate gradients and pass them to the optimizer.
Inherits From: `Optimizer`
Defined in `python/training/sync_replicas_optimizer.py`.
This class is deprecated. For synchrononous training, please use Distribution Strategies.
In a typical asynchronous training environment, it's common to have some stale gradients. For example, with a N-replica asynchronous training, gradients will be applied to the variables N times independently. Depending on each replica's training speed, some gradients might be calculated from copies of the variable from several steps back (N-1 steps on average). This optimizer avoids stale gradients by collecting gradients from all replicas, averaging them, then applying them to the variables in one shot, after which replicas can fetch the new variables and continue.
The following accumulators/queue are created:

- N `gradient accumulators`, one per variable to train. Gradients are pushed to them and the chief worker will wait until enough gradients are collected and then average them before applying to variables. The accumulator will drop all stale gradients (more details in the accumulator op).
- 1 `token` queue where the optimizer pushes the new global_step value after all variables are updated.

The following local variable is created: * `sync_rep_local_step`, one per replica. Compared against the global_step in each accumulator to check for staleness of the gradients.
The optimizer adds nodes to the graph to collect gradients and pause the trainers until variables are updated. For the Parameter Server job:

1. An accumulator is created for each variable, and each replica pushes the gradients into the accumulators instead of directly applying them to the variables.
2. Each accumulator averages once enough gradients (replicas_to_aggregate) have been accumulated.
3. Apply the averaged gradients to the variables.
4. Only after all variables have been updated, increment the global step.
5. Only after step 4, pushes `global_step` in the `token_queue`, once for each worker replica. The workers can now fetch the global step, use it to update its local_step variable and start the next batch. Please note that some workers can consume multiple minibatches, while some may not

consume even one. This is because each worker fetches minibatches as long as a token exists. If one worker is stuck for some reason and does not consume a token, another worker can use it.

For the replicas:

1. Start a step: fetch variables and compute gradients.
2. Once the gradients have been computed, push them into gradient accumulators. Each accumulator will check the staleness and drop the stale.
3. After pushing all the gradients, dequeue an updated value of global_step from the token queue and record that step to its local_step variable. Note that this is effectively a barrier.
4. Start the next batch.

Usage

```
# Create any optimizer to update the variables, say a simple SGD:
opt = GradientDescentOptimizer(learning_rate=0.1)

# Wrap the optimizer with sync_replicas_optimizer with 50 replicas: at each
# step the optimizer collects 50 gradients before applying to variables.
# Note that if you want to have 2 backup replicas, you can change
# total_num_replicas=52 and make sure this number matches how many physical
# replicas you started in your job.
opt = tf.compat.v1.train.SyncReplicasOptimizer(opt, replicas_to_aggregate=50,
                               total_num_replicas=50)

# Some models have startup_delays to help stabilize the model but when using
# sync_replicas training, set it to 0.

# Now you can call `minimize()` or `compute_gradients()` and
# `apply_gradients()` normally
training_op = opt.minimize(total_loss, global_step=self.global_step)


# You can create the hook which handles initialization and queues.
sync_replicas_hook = opt.make_session_run_hook(is_chief)
```

In the training program, every worker will run the train_op as if not synchronized.

```
with training.MonitoredTrainingSession(
    master=workers[worker_id].target, is_chief=is_chief,
    hooks=[sync_replicas_hook]) as mon_sess:
  while not mon_sess.should_stop():
    mon_sess.run(training_op)
```

To use SyncReplicasOptimizer with an `Estimator`, you need to send sync_replicas_hook while calling the fit.

```
my_estimator = DNNClassifier(..., optimizer=opt)
my_estimator.fit(..., hooks=[sync_replicas_hook])
```

`__init__`

```
__init__(
    opt,
    replicas_to_aggregate,
```

```
    total_num_replicas=None,
    variable_averages=None,
    variables_to_average=None,
    use_locking=False,
    name='sync_replicas'
)
```

Construct a sync_replicas optimizer. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: The `SyncReplicaOptimizer` class is deprecated. For synchrononous training, please use Distribution Strategies.

Args:

- `opt`: The actual optimizer that will be used to compute and apply the gradients. Must be one of the Optimizer classes.
- `replicas_to_aggregate`: number of replicas to aggregate for each variable update.
- `total_num_replicas`: Total number of tasks/workers/replicas, could be different from replicas_to_aggregate. If total_num_replicas > replicas_to_aggregate: it is backup_replicas + replicas_to_aggregate. If total_num_replicas < replicas_to_aggregate: Replicas compute multiple batches per update to variables.
- `variable_averages`: Optional `ExponentialMovingAverage` object, used to maintain moving averages for the variables passed in `variables_to_average`.
- `variables_to_average`: a list of variables that need to be averaged. Only needed if variable_averages is passed in.
- `use_locking`: If True use locks for update operation.
- `name`: string. Optional name of the returned operation.

Methods

apply_gradients

```
apply_gradients(
    grads_and_vars,
    global_step=None,
    name=None
)
```

Apply gradients to variables.

This contains most of the synchronization implementation and also wraps the apply_gradients() from the real optimizer.

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:

- `train_op`: The op to dequeue a token so the replicas can exit this batch and start the next one. This is executed by each replica.

Raises:

- `ValueError`: If the grads_and_vars is empty.
- `ValueError`: If global step is not provided, the staleness cannot be checked.

compute_gradients

```
compute_gradients(
    *args,
    **kwargs
)
```

Compute gradients of "loss" for the variables in "var_list".
This simply wraps the compute_gradients() from the real optimizer. The gradients will be aggregated in the apply_gradients() so that user can modify the gradients like clipping with per replica global norm if needed. The global norm with aggregated gradients can be bad as one replica's huge gradients can hurt the gradients from other replicas.

Args:

- **`*args`**: Arguments for compute_gradients().
- **`**kwargs`**: Keyword arguments for compute_gradients().

Returns:
A list of (gradient, variable) pairs.

get_chief_queue_runner

```
get_chief_queue_runner()
```

Returns the QueueRunner for the chief to execute.
This includes the operations to synchronize replicas: aggregate gradients, apply to variables, increment global step, insert tokens to token queue.
Note that this can only be called after calling apply_gradients() which actually generates this queuerunner.

Returns:
A `QueueRunner` for chief to execute.

Raises:

- **`ValueError`**: If this is called before apply_gradients().

get_init_tokens_op

```
get_init_tokens_op(num_tokens=-1)
```

Returns the op to fill the sync_token_queue with the tokens.
This is supposed to be executed in the beginning of the chief/sync thread so that even if the total_num_replicas is less than replicas_to_aggregate, the model can still proceed as the replicas can compute multiple steps per variable update. Make sure: `num_tokens >= replicas_to_aggregate - total_num_replicas`.

Args:

- **`num_tokens`**: Number of tokens to add to the queue.

Returns:
An op for the chief/sync replica to fill the token queue.

Raises:

- **`ValueError`**: If this is called before apply_gradients().
- **`ValueError`**: If num_tokens are smaller than replicas_to_aggregate - total_num_replicas.

get_name

```
get_name()
```

`get_slot`

```
get_slot(
    *args,
    **kwargs
)
```

Return a slot named "name" created for "var" by the Optimizer.
This simply wraps the get_slot() from the actual optimizer.

Args:

- **`*args`**: Arguments for get_slot().
- **`**kwargs`**: Keyword arguments for get_slot().

Returns:
The `Variable` for the slot if it was created, `None` otherwise.

`get_slot_names`

```
get_slot_names(
    *args,
    **kwargs
)
```

Return a list of the names of slots created by the `Optimizer`.
This simply wraps the get_slot_names() from the actual optimizer.

Args:

- **`*args`**: Arguments for get_slot().
- **`**kwargs`**: Keyword arguments for get_slot().

Returns:
A list of strings.

`make_session_run_hook`

```
make_session_run_hook(
    is_chief,
    num_tokens=-1
)
```

Creates a hook to handle SyncReplicasHook ops such as initialization.

`minimize`

```
minimize(
    loss,
    global_step=None,
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=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 or tuple 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.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

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.

Eager Compatibility

When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

`variables`

```
variables()
```

Fetches a list of optimizer variables in the default graph.
This wraps `variables()` from the actual optimizer. It does not include the `SyncReplicasOptimizer`'s local step.

Returns:

A list of variables.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.train.update_checkpoint_state

Updates the content of the 'checkpoint' file. (deprecated)

```
tf.compat.v1.train.update_checkpoint_state(
    save_dir,
    model_checkpoint_path,
    all_model_checkpoint_paths=None,
    latest_filename=None,
    all_model_checkpoint_timestamps=None,
```

```
    last_preserved_timestamp=None
)
```

Defined in `python/training/checkpoint_management.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `tf.train.CheckpointManager` to manage checkpoints rather than manually editing the Checkpoint proto.

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'.
- `all_model_checkpoint_timestamps`: Optional list of timestamps (floats, seconds since the Epoch) indicating when the checkpoints in `all_model_checkpoint_paths` were created.
- `last_preserved_timestamp`: A float, indicating the number of seconds since the Epoch when the last preserved checkpoint was written, e.g. due to a `keep_checkpoint_every_n_hours`parameter (see `tf.contrib.checkpoint.CheckpointManager` for an implementation).

Raises:

- `RuntimeError`: If any of the model checkpoint paths conflict with the file containing CheckpointSate.

tf.compat.v1.train.warm_start

Warm-starts a model using the given settings.

```
tf.compat.v1.train.warm_start(
    ckpt_to_initialize_from,
    vars_to_warm_start='.*',
    var_name_to_vocab_info=None,
    var_name_to_prev_var_name=None
)
```

Defined in `python/training/warm_starting_util.py`.

If you are using a tf.estimator.Estimator, this will automatically be called during training.

Args:

- `ckpt_to_initialize_from`: [Required] A string specifying the directory with checkpoint file(s) or path to checkpoint from which to warm-start the model parameters.
- `vars_to_warm_start`: [Optional] One of the following:
- A regular expression (string) that captures which variables to warm-start (see tf.compat.v1.get_collection). This expression will only consider variables in the TRAINABLE_VARIABLES collection -- if you need to warm-start non_TRAINABLE vars (such as optimizer accumulators or batch norm statistics), please use the below option.
- A list of strings, each a regex scope provided to tf.compat.v1.get_collection with GLOBAL_VARIABLES (please see tf.compat.v1.get_collection). For backwards compatibility reasons, this is separate from the single-string argument type.
- A list of Variables to warm-start. If you do not have access to the `Variable` objects at the call site, please use the above option.
- `None`, in which case only TRAINABLE variables specified in `var_name_to_vocab_info`will be warm-started.

Defaults to `'.*'`, which warm-starts all variables in the TRAINABLE_VARIABLES collection. Note that this excludes variables such as accumulators and moving statistics from batch norm.

- **`var_name_to_vocab_info`**: [Optional] Dict of variable names (strings) to`tf.estimator.VocabInfo`. The variable names should be "full" variables, not the names of the partitions. If not explicitly provided, the variable is assumed to have no (changes to) vocabulary.
- **`var_name_to_prev_var_name`**: [Optional] Dict of variable names (strings) to name of the previously-trained variable in `ckpt_to_initialize_from`. If not explicitly provided, the name of the variable is assumed to be same between previous checkpoint and current model. Note that this has no effect on the set of variables that is warm-started, and only controls name mapping (use `vars_to_warm_start` for controlling what variables to warm-start).

Raises:

- **`ValueError`**: If the WarmStartSettings contains prev_var_name or VocabInfo configuration for variable names that are not used. This is to ensure a stronger check for variable configuration than relying on users to examine the logs.

tf.compat.v1.train.WorkerSessionCreator

- **Contents**
- Class WorkerSessionCreator
- __init__
- Methods
  - create_session

Class `WorkerSessionCreator`

Creates a tf.compat.v1.Session for a worker.

Inherits From: `SessionCreator`

Defined in `python/training/monitored_session.py`.

`__init__`

```
__init__(
    scaffold=None,
    master='',
    config=None,
    max_wait_secs=(30 * 60)
)
```

Initializes a worker session creator.

Args:

- **`scaffold`**: A `Scaffold` used for gathering or building supportive ops. If not specified a default one is created. It's used to finalize the graph.
- **`master`**: `String` representation of the TensorFlow master to use.
- **`config`**: `ConfigProto` proto used to configure the session.
- **`max_wait_secs`**: Maximum time to wait for the session to become available.

Methods

`create_session`

```
create_session()
```

Module: tf.compat.v1.train.experimental

- **Contents**
- Classes

- Functions

Public API for tf.train.experimental namespace.

Classes

`class DynamicLossScale`: Loss scale that dynamically adjusts itself.
`class FixedLossScale`: Loss scale with a fixed value.
`class LossScale`: Loss scale base class.
`class MixedPrecisionLossScaleOptimizer`: An optimizer that applies loss scaling.
`class PythonState`: A mixin for putting Python state in an object-based checkpoint.

Functions

`disable_mixed_precision_graph_rewrite(...)`: Disables the mixed precision graph rewrite.
`enable_mixed_precision_graph_rewrite(...)`: Enable mixed precision via a graph rewrite.

tf.compat.v1.train.experimental.disable_mixed_precision_graph_rewrite

Disables the mixed precision graph rewrite.

```
tf.compat.v1.train.experimental.disable_mixed_precision_graph_rewrite()
```

Defined in `python/training/experimental/mixed_precision.py`.

After this is called, the mixed precision graph rewrite will no longer run for new Sessions, and so float32 operations will no longer be converted to float16 in such Sessions. However, any existing Sessions will continue to have the graph rewrite enabled if they were created after`enable_mixed_precision_graph_rewrite` was called but before`disable_mixed_precision_graph_rewrite` was called.

This does not undo the effects of loss scaling. Any optimizers wrapped with a LossScaleOptimizer will continue to do loss scaling, although this loss scaling will no longer be useful if the optimizer is used in new Sessions, as the graph rewrite no longer converts the graph to use float16.

This function is useful for unit testing. A unit tests can test using the mixed precision graph rewrite, then disable it so future unit tests continue using float32. If this is done, unit tests should not share a single session, as `enable_mixed_precision_graph_rewrite` and`disable_mixed_precision_graph_rewrite` have no effect on existing sessions.

tf.compat.v1.train.experimental.enable_mixed_precision_graph_rewrite

Enable mixed precision via a graph rewrite.

```
tf.compat.v1.train.experimental.enable_mixed_precision_graph_rewrite(
    opt,
    loss_scale='dynamic'
)
```

Defined in `python/training/experimental/mixed_precision.py`.

Mixed precision is the use of both float16 and float32 when training a model, and is used to make the model run faster. This function will use mixed precision to speed up the execution time of your model when run on a GPU. It does this by changing the dtype of certain operations in the graph from float32 to float16.

This function additionally wraps an Optimizer with a LossScaleOptimizer, which is required to prevent underflow in the float16 tensors during the backwards pass. An optimizer must be passed to this function, which will then be wrapped to use loss scaling.

When this function is used, gradients should only be computed and applied with the returned optimizer, either by calling `opt.minimize()` or `opt.compute_gradients()` followed by `opt.apply_gradients()`. Gradients should not be computed with `tf.gradients` or `tf.GradientTape`. This is because the returned optimizer will apply loss scaling, and`tf.gradients`/`tf.GradientTape` will not. If you do directly use `tf.gradients` or `tf.GradientTape`, your model may train to a worse quality.
When eager execution is enabled, the mixed precision graph rewrite is only enabled within `tf.function`s, as outside `tf.function`s, there is no graph.
When enabled, mixed precision is only used on Volta GPUs and above. The parts of the graph on CPUs and TPUs are untouched by the graph rewrite.

Args:

- **`opt`**: An instance of a `tf.keras.optimizers.Optimizer` or a `tf.train.Optimizer`.
- **`loss_scale`**: Either an int/float, the string "dynamic", or an instance of a`tf.train.experimental.LossScale`. The loss scale to use. It is recommended to keep this as its default value of "dynamic".

Returns:

A version of `opt` that will use loss scaling to prevent underflow.

Module: tf.train

- **Contents**
- Modules
- Classes
- Functions

Support for training models.
See the Training guide.

Modules

`experimental` module: Public API for tf.train.experimental namespace.

Classes

`class BytesList`
`class Checkpoint`: Groups trackable objects, saving and restoring them.
`class CheckpointManager`: Deletes old checkpoints.
`class ClusterDef`
`class ClusterSpec`: Represents a cluster as a set of "tasks", organized into "jobs".
`class Coordinator`: A coordinator for threads.
`class Example`
`class ExponentialMovingAverage`: Maintains moving averages of variables by employing an exponential decay.
`class Feature`
`class FeatureList`
`class FeatureLists`
`class Features`
`class FloatList`
`class Int64List`
`class JobDef`
`class SequenceExample`
`class ServerDef`

Functions

`checkpoints_iterator(...)`: Continuously yield new checkpoint files as they appear.
`get_checkpoint_state(...)`: Returns CheckpointState proto from the "checkpoint" file.
`latest_checkpoint(...)`: Finds the filename of latest saved checkpoint file.

`list_variables(...)`: Returns list of all variables in the checkpoint.
`load_checkpoint(...)`: Returns `CheckpointReader` for checkpoint found in `ckpt_dir_or_file`.
`load_variable(...)`: Returns the tensor value of the given variable in the checkpoint.

tf.train.BytesList

- **Contents**
- Class BytesList
  - Aliases:
  - Used in the tutorials:
- Properties
  - value

Class `BytesList`

Aliases:

- Class `tf.compat.v1.train.BytesList`
- Class `tf.compat.v2.train.BytesList`
- Class `tf.train.BytesList`

Defined in `core/example/feature.proto`.

Used in the tutorials:

- Using TFRecords and tf.Example

Properties

`value`

`repeated bytes value`

tf.train.Checkpoint

- **Contents**
- Class Checkpoint
  - Aliases:
  - Used in the guide:
  - Used in the tutorials:

Class `Checkpoint`

Groups trackable objects, saving and restoring them.

Aliases:

- Class `tf.compat.v2.train.Checkpoint`
- Class `tf.train.Checkpoint`

Defined in `python/training/tracking/util.py`.

Used in the guide:

- Eager essentials
- Training checkpoints

Used in the tutorials:

- Deep Convolutional Generative Adversarial Network
- Image Captioning with Attention
- Neural Machine Translation with Attention
- Pix2Pix
- Transformer model for language understanding

- [tf.distribute.Strategy with training loops]

`Checkpoint`'s constructor accepts keyword arguments whose values are types that contain trackable state, such as `tf.keras.optimizers.Optimizer` implementations, `tf.Variable`, `tf.keras.Layer` implementations, or `tf.keras.Model` implementations. It saves these values with a checkpoint, and maintains a `save_counter` for numbering checkpoints.

Example usage:

```
import tensorflow as tf
import os

checkpoint_directory = "/tmp/training_checkpoints"
checkpoint_prefix = os.path.join(checkpoint_directory, "ckpt")

checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
status = checkpoint.restore(tf.train.latest_checkpoint(checkpoint_directory))
for _ in range(num_training_steps):
  optimizer.minimize( ... )  # Variables will be restored on creation.
status.assert_consumed()  # Optional sanity checks.
checkpoint.save(file_prefix=checkpoint_prefix)
```

`Checkpoint.save` and `Checkpoint.restore` write and read object-based checkpoints, in contrast to TensorFlow 1.x's `tf.compat.v1.train.Saver` which writes and reads `variable.name` based checkpoints. Object-based checkpointing saves a graph of dependencies between Python objects (`Layer`s, `Optimizer`s, `Variable`s, etc.) with named edges, and this graph is used to match variables when restoring a checkpoint. It can be more robust to changes in the Python program, and helps to support restore-on-create for variables.

`Checkpoint` objects have dependencies on the objects passed as keyword arguments to their constructors, and each dependency is given a name that is identical to the name of the keyword argument for which it was created. TensorFlow classes like `Layer`s and `Optimizer`s will automatically add dependencies on their variables (e.g. "kernel" and "bias" for`tf.keras.layers.Dense`). Inheriting from `tf.keras.Model` makes managing dependencies easy in user-defined classes, since `Model` hooks into attribute assignment. For example:

```
class Regress(tf.keras.Model):

  def __init__(self):
    super(Regress, self).__init__()
    self.input_transform = tf.keras.layers.Dense(10)
    # ...

  def call(self, inputs):
    x = self.input_transform(inputs)
    # ...
```

This `Model` has a dependency named "input_transform" on its `Dense` layer, which in turn depends on its variables. As a result, saving an instance of `Regress` using `tf.train.Checkpoint` will also save all the variables created by the `Dense` layer.

When variables are assigned to multiple workers, each worker writes its own section of the checkpoint. These sections are then merged/re-indexed to behave as a single checkpoint. This avoids copying all variables to one worker, but does require that all workers see a common filesystem.

While `tf.keras.Model.save_weights` and `tf.train.Checkpoint.save` save in the same format, note that the root of the resulting checkpoint is the object the save method is attached to. This means saving a `tf.keras.Model` using `save_weights` and loading into a `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not match the `Model`'s variables. See the guide to training checkpoints for details.
Prefer `tf.train.Checkpoint` over `tf.keras.Model.save_weights` for training checkpoints.

Attributes:

- **`save_counter`**: Incremented when `save()` is called. Used to number checkpoints.

`__init__`

```
__init__(**kwargs)
```

Group objects into a training checkpoint.

Args:

- **`**kwargs`**: Keyword arguments are set as attributes of this object, and are saved with the checkpoint. Values must be trackable objects.

Raises:

- **`ValueError`**: If objects in `kwargs` are not trackable.

Properties

`save_counter`

An integer variable which starts at zero and is incremented on save.
Used to number checkpoints.

Returns:

The save counter variable.

Methods

`restore`

```
restore(save_path)
```

Restore a training checkpoint.
Restores this `Checkpoint` and any objects it depends on.
Either assigns values immediately if variables to restore have been created already, or defers restoration until the variables are created. Dependencies added after this call will be matched if they have a corresponding object in the checkpoint (the restore request will queue in any trackable object waiting for the expected dependency to be added).
To ensure that loading is complete and no more assignments will take place, use the `assert_consumed()` method of the status object returned by `restore`:

```
checkpoint = tf.train.Checkpoint( ... )
checkpoint.restore(path).assert_consumed()
```

An exception will be raised if any Python objects in the dependency graph were not found in the checkpoint, or if any checkpointed values do not have a matching Python object.
Name-based `tf.compat.v1.train.Saver` checkpoints from TensorFlow 1.x can be loaded using this method. Names are used to match variables. Re-encode name-based checkpoints using `tf.train.Checkpoint.save` as soon as possible.

Args:

- **`save_path`**: The path to the checkpoint, as returned by `save` or`tf.train.latest_checkpoint`. If None (as when there is no latest checkpoint for `tf.train.latest_checkpoint` to return), returns an object which may run initializers for objects in the dependency graph. If the checkpoint was written by the name-based`tf.compat.v1.train.Saver`, names are used to match variables.

Returns:

A load status object, which can be used to make assertions about the status of a checkpoint restoration.

The returned status object has the following methods:

- `assert_consumed()`: Raises an exception if any variables/objects are unmatched: either checkpointed values which don't have a matching Python object or Python objects in the dependency graph with no values in the checkpoint. This method returns the status object, and so may be chained with other assertions.
- `assert_existing_objects_matched()`: Raises an exception if any existing Python objects in the dependency graph are unmatched. Unlike `assert_consumed`, this assertion will pass if values in the checkpoint have no corresponding Python objects. For example a `tf.keras.Layer` object which has not yet been built, and so has not created any variables, will pass this assertion but fail `assert_consumed`. Useful when loading part of a larger checkpoint into a new Python program, e.g. a training checkpoint with a `tf.compat.v1.train.Optimizer`was saved but only the state required for inference is being loaded. This method returns the status object, and so may be chained with other assertions.
- `assert_nontrivial_match()`: Asserts that something aside from the root object was matched. This is a very weak assertion, but is useful for sanity checking in library code where objects may exist in the checkpoint which haven't been created in Python and some Python objects may not have a checkpointed value.
- `expect_partial()`: Silence warnings about incomplete checkpoint restores. Warnings are otherwise printed for unused parts of the checkpoint file or object when the `Checkpoint` object is deleted (often at program shutdown).

`save`

```
save(file_prefix)
```

Saves a training checkpoint and provides basic checkpoint management.

The saved checkpoint includes variables created by this object and any trackable objects it depends on at the time `Checkpoint.save()` is called.

`save` is a basic convenience wrapper around the `write` method, sequentially numbering checkpoints using `save_counter` and updating the metadata used by `tf.train.latest_checkpoint`. More advanced checkpoint management, for example garbage collection and custom numbering, may be provided by other utilities which also wrap `write` (`tf.contrib.checkpoint.CheckpointManager`for example).

Args:

- **`file_prefix`**: A prefix to use for the checkpoint filenames (/path/to/directory/and_a_prefix). Names are generated based on this prefix and `Checkpoint.save_counter`.

Returns:

The full path to the checkpoint.

`write`

```
write(file_prefix)
```

Writes a training checkpoint.

The checkpoint includes variables created by this object and any trackable objects it depends on at the time `Checkpoint.write()` is called.

`write` does not number checkpoints, increment `save_counter`, or update the metadata used by `tf.train.latest_checkpoint`. It is primarily intended for use by higher level checkpoint management utilities. `save` provides a very basic implementation of these features.

Args:

- **`file_prefix`**: A prefix to use for the checkpoint filenames (/path/to/directory/and_a_prefix).

Returns:

The full path to the checkpoint (i.e. `file_prefix`).

tf.train.CheckpointManager

- **Contents**
- Class CheckpointManager
  - Aliases:
  - Used in the guide:
  - Used in the tutorials:

Class `CheckpointManager`

Deletes old checkpoints.

Aliases:

- Class `tf.compat.v1.train.CheckpointManager`
- Class `tf.compat.v2.train.CheckpointManager`
- Class `tf.train.CheckpointManager`

Defined in `python/training/checkpoint_management.py`.

Used in the guide:

- Training checkpoints

Used in the tutorials:

- Image Captioning with Attention
- Transformer model for language understanding

Example usage:

```
import tensorflow as tf
checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
manager = tf.contrib.checkpoint.CheckpointManager(
    checkpoint, directory="/tmp/model", max_to_keep=5)
status = checkpoint.restore(manager.latest_checkpoint)
while True:
  # train
  manager.save()
```

`CheckpointManager` preserves its own state across instantiations (see the `__init__` documentation for details). Only one should be active in a particular directory at a time.

`__init__`

```
__init__(
    checkpoint,
    directory,
    max_to_keep,
    keep_checkpoint_every_n_hours=None,
    checkpoint_name='ckpt'
```

```
)
```

Configure a `CheckpointManager` for use in `directory`.
If a `CheckpointManager` was previously used in `directory`, its state will be restored. This includes the list of managed checkpoints and the timestamp bookkeeping necessary to support`keep_checkpoint_every_n_hours`. The behavior of the new `CheckpointManager` will be the same as the previous `CheckpointManager`, including cleaning up existing checkpoints if appropriate.
Checkpoints are only considered for deletion just after a new checkpoint has been added. At that point, `max_to_keep` checkpoints will remain in an "active set". Once a checkpoint is preserved by`keep_checkpoint_every_n_hours` it will not be deleted by this `CheckpointManager` or any future `CheckpointManager` instantiated in `directory` (regardless of the new setting of`keep_checkpoint_every_n_hours`). The `max_to_keep` checkpoints in the active set may be deleted by this `CheckpointManager` or a future `CheckpointManager` instantiated in `directory` (subject to its `max_to_keep` and `keep_checkpoint_every_n_hours` settings).

Args:

- **`checkpoint`**: The `tf.train.Checkpoint` instance to save and manage checkpoints for.
- **`directory`**: The path to a directory in which to write checkpoints. A special file named "checkpoint" is also written to this directory (in a human-readable text format) which contains the state of the `CheckpointManager`.
- **`max_to_keep`**: An integer, the number of checkpoints to keep. Unless preserved by `keep_checkpoint_every_n_hours`, checkpoints will be deleted from the active set, oldest first, until only `max_to_keep` checkpoints remain. If `None`, no checkpoints are deleted and everything stays in the active set. Note that `max_to_keep=None` will keep all checkpoint paths in memory and in the checkpoint state protocol buffer on disk.
- **`keep_checkpoint_every_n_hours`**: Upon removal from the active set, a checkpoint will be preserved if it has been at least `keep_checkpoint_every_n_hours` since the last preserved checkpoint. The default setting of `None` does not preserve any checkpoints in this way.
- **`checkpoint_name`**: Custom name for the checkpoint file.

Raises:

- **`ValueError`**: If `max_to_keep` is not a positive integer.

Properties

`checkpoints`

A list of managed checkpoints.
Note that checkpoints saved due to `keep_checkpoint_every_n_hours` will not show up in this list (to avoid ever-growing filename lists).

Returns:
A list of filenames, sorted from oldest to newest.

`latest_checkpoint`

The prefix of the most recent checkpoint in `directory`.
Equivalent to `tf.train.latest_checkpoint(directory)` where `directory` is the constructor argument to `CheckpointManager`.
Suitable for passing to `tf.train.Checkpoint.restore` to resume training.

Returns:
The checkpoint prefix. If there are no checkpoints, returns `None`.

Methods

`save`

```
save(checkpoint_number=None)
```

Creates a new checkpoint and manages it.

Args:

- **`checkpoint_number`**: An optional integer, or an integer-dtype `Variable` or `Tensor`, used to number the checkpoint. If `None` (default), checkpoints are numbered using `checkpoint.save_counter`. Even if `checkpoint_number` is provided, `save_counter` is still incremented. A user-provided `checkpoint_number` is not incremented even if it is a `Variable`.

Returns:

The path to the new checkpoint. It is also recorded in the `checkpoints` and `latest_checkpoint`properties.

tf.train.checkpoints_iterator

- **Contents**
- Aliases:

Continuously yield new checkpoint files as they appear.

Aliases:

- `tf.compat.v1.train.checkpoints_iterator`
- `tf.compat.v2.train.checkpoints_iterator`
- `tf.train.checkpoints_iterator`

```
tf.train.checkpoints_iterator(
    checkpoint_dir,
    min_interval_secs=0,
    timeout=None,
    timeout_fn=None
)
```

Defined in `python/training/checkpoint_utils.py`.

The iterator only checks for new checkpoints when control flow has been reverted to it. This means it can miss checkpoints if your code takes longer to run between iterations than `min_interval_secs` or the interval at which new checkpoints are written.

The `timeout` argument is the maximum number of seconds to block waiting for a new checkpoint. It is used in combination with the `timeout_fn` as follows:

- If the timeout expires and no `timeout_fn` was specified, the iterator stops yielding.
- If a `timeout_fn` was specified, that function is called and if it returns a true boolean value the iterator stops yielding.
- If the function returns a false boolean value then the iterator resumes the wait for new checkpoints. At this point the timeout logic applies again.

This behavior gives control to callers on what to do if checkpoints do not come fast enough or stop being generated. For example, if callers have a way to detect that the training has stopped and know that no new checkpoints will be generated, they can provide a `timeout_fn` that returns `True` when the training has stopped. If they know that the training is still going on they return `False` instead.

Args:

- **`checkpoint_dir`**: The directory in which checkpoints are saved.
- **`min_interval_secs`**: The minimum number of seconds between yielding checkpoints.

- `timeout`: The maximum number of seconds to wait between checkpoints. If left as `None`, then the process will wait indefinitely.
- `timeout_fn`: Optional function to call after a timeout. If the function returns True, then it means that no new checkpoints will be generated and the iterator will exit. The function is called with no arguments.

Yields:
String paths to latest checkpoint files as they arrive.

tf.train.ClusterDef

Class `ClusterDef`

Aliases:

- Class `tf.compat.v1.train.ClusterDef`
- Class `tf.compat.v2.train.ClusterDef`
- Class `tf.train.ClusterDef`

Defined in `core/protobuf/cluster.proto`.

Properties

`job`

`repeated JobDef job`

tf.train.ClusterSpec

Class `ClusterSpec`

Represents a cluster as a set of "tasks", organized into "jobs".

Aliases:

- Class `tf.compat.v1.train.ClusterSpec`
- Class `tf.compat.v2.train.ClusterSpec`
- Class `tf.train.ClusterSpec`

Defined in `python/training/server_lib.py`.

A `tf.train.ClusterSpec` represents the set of processes that participate in a distributed TensorFlow computation. Every `tf.distribute.Server` is constructed in a particular cluster.

To create a cluster with two jobs and five tasks, you specify the mapping from job names to lists of network addresses (typically hostname-port pairs).

```
cluster = tf.train.ClusterSpec({"worker": ["worker0.example.com:2222",
                                           "worker1.example.com:2222",
                                           "worker2.example.com:2222"],
                                "ps": ["ps0.example.com:2222",
```

```
                                            "ps1.example.com:2222"]})
```

Each job may also be specified as a sparse mapping from task indices to network addresses. This enables a server to be configured without needing to know the identity of (for example) all other worker tasks:

```
cluster = tf.train.ClusterSpec({"worker": {1: "worker1.example.com:2222"},
                                    "ps": ["ps0.example.com:2222",
                                           "ps1.example.com:2222"]})
```

`__init__`

```
__init__(cluster)
```

Creates a `ClusterSpec`.

Args:

- **`cluster`**: A dictionary mapping one or more job names to (i) a list of network addresses, or (ii) a dictionary mapping integer task indices to network addresses; or a `tf.train.ClusterDef`protocol buffer.

Raises:

- **`TypeError`**: If `cluster` is not a dictionary mapping strings to lists of strings, and not a `tf.train.ClusterDef` protobuf.

Properties

`jobs`

Returns a list of job names in this cluster.

Returns:

A list of strings, corresponding to the names of jobs in this cluster.

Methods

`__bool__`

```
__bool__()
```

`__eq__`

```
__eq__(other)
```

`__ne__`

```
__ne__(other)
```

`__nonzero__`

```
__nonzero__()
```

`as_cluster_def`

```
as_cluster_def()
```

Returns a `tf.train.ClusterDef` protocol buffer based on this cluster.

`as_dict`

```
as_dict()
```

Returns a dictionary from job names to their tasks.
For each job, if the task index space is dense, the corresponding value will be a list of network addresses; otherwise it will be a dictionary mapping (sparse) task indices to the corresponding addresses.

Returns:
A dictionary mapping job names to lists or dictionaries describing the tasks in those jobs.

`job_tasks`

```
job_tasks(job_name)
```

Returns a mapping from task ID to address in the given job.
NOTE: For backwards compatibility, this method returns a list. If the given job was defined with a sparse set of task indices, the length of this list may not reflect the number of tasks defined in this job. Use the `tf.train.ClusterSpec.num_tasks` method to find the number of tasks defined in a particular job.

Args:

- **`job_name`**: The string name of a job in this cluster.

Returns:
A list of task addresses, where the index in the list corresponds to the task index of each task. The list may contain `None` if the job was defined with a sparse set of task indices.

Raises:

- **`ValueError`**: If `job_name` does not name a job in this cluster.

`num_tasks`

```
num_tasks(job_name)
```

Returns the number of tasks defined in the given job.

Args:

- **`job_name`**: The string name of a job in this cluster.

Returns:
The number of tasks defined in the given job.

Raises:

- **`ValueError`**: If `job_name` does not name a job in this cluster.

`task_address`

```
task_address(
    job_name,
    task_index
)
```

Returns the address of the given task in the given job.

Args:

- **`job_name`**: The string name of a job in this cluster.
- **`task_index`**: A non-negative integer.

Returns:
The address of the given task in the given job.

Raises:

- **`ValueError`**: If `job_name` does not name a job in this cluster, or no task with index `task_index` is defined in that job.

`task_indices`

```
task_indices(job_name)
```

Returns a list of valid task indices in the given job.

Args:

- **`job_name`**: The string name of a job in this cluster.

Returns:
A list of valid task indices in the given job.

Raises:

- **`ValueError`**: If `job_name` does not name a job in this cluster, or no task with index `task_index` is defined in that job.

tf.train.Coordinator

- **Contents**
- Class Coordinator
  - Aliases:
- __init__
- Properties

Class `Coordinator`

A coordinator for threads.

Aliases:

- Class `tf.compat.v1.train.Coordinator`
- Class `tf.compat.v2.train.Coordinator`
- Class `tf.train.Coordinator`

Defined in `python/training/coordinator.py`.

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 `request_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 RuntimeError 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 RuntimeError:
  ...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()
```

`__init__`

```
__init__(clean_stop_exception_types=None)
```

Create a new Coordinator.

Args:

- **`clean_stop_exception_types`**: Optional tuple of Exception types that should cause a clean stop of the coordinator. If an exception of one of these types is reported to `request_stop(ex)` the coordinator will behave as if `request_stop(None)` was called. Defaults to `(tf.errors.OutOfRangeError,)` which is used by input queues to signal the end of input. When feeding training data from a Python iterator it is common to add `StopIteration` to this list.

Properties

`joined`

Methods

`clear_stop`

```
clear_stop()
```

Clears the stop flag.
After this is called, calls to `should_stop()` will return `False`.

`join`

```
join(
    threads=None,
    stop_grace_period_secs=120,
    ignore_live_threads=False
)
```

Wait for threads to terminate.
This call blocks until a set of threads have terminated. The set of thread is the union of the threads passed in the `threads` argument and the list of threads that registered with the coordinator by calling `Coordinator.register_thread()`.
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 in addition to the registered threads.
- **`stop_grace_period_secs`**: Number of seconds given to threads to stop after `request_stop()` has been called.
- **`ignore_live_threads`**: If `False`, raises an error if any of the threads are still alive after `stop_grace_period_secs`.

Raises:

- **`RuntimeError`**: If any thread is still alive after `request_stop()` is called and the grace period expires.

`raise_requested_exception`

```
raise_requested_exception()
```

If an exception has been passed to `request_stop`, this raises it.

`register_thread`

```
register_thread(thread)
```

Register a thread to join.

Args:

- **`thread`**: A Python thread to join.

`request_stop`

```
request_stop(ex=None)
```

Request that the threads stop.
After this is called, calls to `should_stop()` will return `True`.
Note: If an exception is being passed in, in must be in the context of handling the exception (i.e. **`try: ... except Exception as ex: ...`**) and not a newly created one.

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()`.

`should_stop`

```
should_stop()
```

Check if stop was requested.

Returns:
True if a stop was requested.

`stop_on_exception`

```
stop_on_exception(
    *args,
    **kwds
)
```

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...
```

```
except:
  coord.request_stop(sys.exc_info())
```

Yields:
nothing.

`wait_for_stop`

```
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.

tf.train.Example

- **Contents**
- Class Example
  - Aliases:
  - Used in the guide:
  - Used in the tutorials:
- Properties
  - features

Class `Example`

Aliases:

- Class `tf.compat.v1.train.Example`
- Class `tf.compat.v2.train.Example`
- Class `tf.train.Example`

Defined in `core/example/example.proto`.

Used in the guide:

- Using the SavedModel format

Used in the tutorials:

- Using TFRecords and tf.Example

Properties

`features`

`Features features`

tf.train.ExponentialMovingAverage

- **Contents**
- Class ExponentialMovingAverage
  - Aliases:
- __init__
- Properties

Class `ExponentialMovingAverage`

Maintains moving averages of variables by employing an exponential decay.

Aliases:

- Class `tf.compat.v1.train.ExponentialMovingAverage`
- Class `tf.compat.v2.train.ExponentialMovingAverage`
- Class `tf.train.ExponentialMovingAverage`

Defined in `python/training/moving_averages.py`.

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)

with tf.control_dependencies([opt_op]):
    # Create the shadow variables, and add ops to maintain moving averages
    # of var0 and var1. This also creates 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.
    training_op = ema.apply([var0, var1])

...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 `tf.compat.v1.train.Saver` 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.compat.v1.train.Saver({shadow_var0_name: var0, shadow_var1_name:
var1})
saver.restore(...checkpoint filename...)
# var0 and var1 now hold the moving average values
```

`__init__`

```
__init__(
    decay,
    num_updates=None,
    zero_debias=False,
    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.
- **`zero_debias`**: If `True`, zero debias moving-averages that are initialized with tensors.
- **`name`**: String. Optional prefix name to use for the name of ops added in `apply()`.

Properties

`name`

The name of this ExponentialMovingAverage object.

Methods

`apply`

```
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 and zero debiased (see docstring in `assign_moving_average` for more details).

shadow variables are created with `trainable=False` and added to the `GraphKeys.ALL_VARIABLES`collection. They will be returned by calls to `tf.compat.v1.global_variables()`.
Returns an op that updates all shadow variables from the current value of their associated variables.
Note that `apply()` can be called multiple times. When eager execution is enabled each call to apply will update the variables once, so this needs to be called in a loop.

Args:

- **`var_list`**: A list of Variable or Tensor objects. The variables and Tensors must be of types bfloat16, float16, float32, or float64.

Returns:

An Operation that updates the moving averages.

Raises:

- **`TypeError`**: If the arguments are not an allowed type.

average

```
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.

average_name

```
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.compat.v1.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`.

variables_to_restore

```
variables_to_restore(moving_avg_variables=None)
```

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.compat.v1.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
```

Args:

- **`moving_avg_variables`**: a list of variables that require to use of the moving average variable name to be restored. If None, it will default to variables.moving_average_variables() + variables.trainable_variables()

Returns:

A map from restore_names to variables. The restore_name is either the original or the moving average version of the variable name, depending on whether the variable name is in the `moving_avg_variables`.

tf.train.Feature

Class `Feature`

Aliases:

- Class `tf.compat.v1.train.Feature`
- Class `tf.compat.v2.train.Feature`
- Class `tf.train.Feature`

Defined in `core/example/feature.proto`.

Used in the tutorials:

- Using TFRecords and tf.Example

Properties

`bytes_list`

`BytesList bytes_list`

`float_list`

`FloatList float_list`

`int64_list`

`Int64List int64_list`

tf.train.FeatureList

Class `FeatureList`

Aliases:

- Class `tf.compat.v1.train.FeatureList`

- Class `tf.compat.v2.train.FeatureList`
- Class `tf.train.FeatureList`

Defined in `core/example/feature.proto`.

Properties

`feature`

`repeated Feature feature`

tf.train.FeatureLists

- **Contents**
- Class FeatureLists
  - Aliases:
- Child Classes
- Properties
  - feature_list

Class `FeatureLists`

Aliases:

- Class `tf.compat.v1.train.FeatureLists`
- Class `tf.compat.v2.train.FeatureLists`
- Class `tf.train.FeatureLists`

Defined in `core/example/feature.proto`.

Child Classes

`class FeatureListEntry`

Properties

`feature_list`

`repeated FeatureListEntry feature_list`

tf.train.FeatureLists.FeatureListEntry

- **Contents**
- Class FeatureListEntry
  - Aliases:
- Properties
  - key
  - value

Class `FeatureListEntry`

Aliases:

- Class `tf.compat.v1.train.FeatureLists.FeatureListEntry`
- Class `tf.compat.v2.train.FeatureLists.FeatureListEntry`
- Class `tf.train.FeatureLists.FeatureListEntry`

Defined in `core/example/feature.proto`.

Properties

`key`

`string key`

`value`

`FeatureList value`

tf.train.Features

- **Contents**
- Class Features
  - Aliases:
  - Used in the tutorials:
- Child Classes
- Properties
  - feature

Class `Features`

Aliases:

- Class `tf.compat.v1.train.Features`
- Class `tf.compat.v2.train.Features`
- Class `tf.train.Features`

Defined in `core/example/feature.proto`.

Used in the tutorials:

- Using TFRecords and tf.Example

Child Classes

`class FeatureEntry`

Properties

`feature`

`repeated FeatureEntry feature`

tf.train.Features.FeatureEntry

- **Contents**
- Class FeatureEntry
  - Aliases:
- Properties
  - key
  - value

Class `FeatureEntry`

Aliases:

- Class `tf.compat.v1.train.Features.FeatureEntry`
- Class `tf.compat.v2.train.Features.FeatureEntry`
- Class `tf.train.Features.FeatureEntry`

Defined in `core/example/feature.proto`.

Properties

`key`

`string key`

`value`

`Feature value`

tf.train.FloatList

- **Contents**
- Class FloatList
  - Aliases:
  - Used in the tutorials:
- Properties
  - value

Class `FloatList`

Aliases:

- Class `tf.compat.v1.train.FloatList`
- Class `tf.compat.v2.train.FloatList`
- Class `tf.train.FloatList`

Defined in `core/example/feature.proto`.

Used in the tutorials:

- Using TFRecords and tf.Example

Properties

`value`

`repeated float value`

tf.train.get_checkpoint_state

- **Contents**
- Aliases:

Returns CheckpointState proto from the "checkpoint" file.

Aliases:

- `tf.compat.v1.train.get_checkpoint_state`
- `tf.compat.v2.train.get_checkpoint_state`
- `tf.train.get_checkpoint_state`

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

Defined in `python/training/checkpoint_management.py`.

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.

Raises:

- **`ValueError`**: if the checkpoint read doesn't have model_checkpoint_path set.

tf.train.Int64List

- **Contents**

Class `Int64List`

Aliases:

- Class `tf.compat.v1.train.Int64List`
- Class `tf.compat.v2.train.Int64List`
- Class `tf.train.Int64List`

Defined in `core/example/feature.proto`.

Used in the tutorials:

- Using TFRecords and tf.Example

Properties

`value`

`repeated int64 value`

tf.train.JobDef

Contents

Class `JobDef`

Aliases:

- Class `tf.compat.v1.train.JobDef`
- Class `tf.compat.v2.train.JobDef`
- Class `tf.train.JobDef`

Defined in `core/protobuf/cluster.proto`.

Child Classes

`class TasksEntry`

Properties

`name`

`string name`

`tasks`

`repeated TasksEntry tasks`

tf.train.JobDef.TasksEntry

Contents

- Properties
  - key
  - value

Class `TasksEntry`

Aliases:

- Class `tf.compat.v1.train.JobDef.TasksEntry`
- Class `tf.compat.v2.train.JobDef.TasksEntry`
- Class `tf.train.JobDef.TasksEntry`

Defined in `core/protobuf/cluster.proto`.

Properties

`key`

`int32 key`

`value`

`string value`

tf.train.latest_checkpoint

- **Contents**
- Aliases:
- Used in the guide:
- Used in the tutorials:

Finds the filename of latest saved checkpoint file.

Aliases:

- `tf.compat.v1.train.latest_checkpoint`
- `tf.compat.v2.train.latest_checkpoint`
- `tf.train.latest_checkpoint`

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

Defined in `python/training/checkpoint_management.py`.

Used in the guide:

- Eager essentials
- Training checkpoints

Used in the tutorials:

- Deep Convolutional Generative Adversarial Network
- Distributed training with Keras
- Neural Machine Translation with Attention
- Pix2Pix
- Save and restore models
- Text generation with an RNN
- tf.distribute.Strategy with training loops

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.list_variables

- **Contents**
- Aliases:
- Used in the guide:

Returns list of all variables in the checkpoint.

Aliases:

- `tf.compat.v1.train.list_variables`
- `tf.compat.v2.train.list_variables`
- `tf.train.list_variables`

```
tf.train.list_variables(ckpt_dir_or_file)

```

Defined in `python/training/checkpoint_utils.py`.

Used in the guide:

- Training checkpoints

Args:

- **`ckpt_dir_or_file`**: Directory with checkpoints file or path to checkpoint.

Returns:
List of tuples `(name, shape)`.

tf.train.load_checkpoint

- **Contents**
- Aliases:

Returns `CheckpointReader` for checkpoint found in `ckpt_dir_or_file`.

Aliases:

- `tf.compat.v1.train.load_checkpoint`
- `tf.compat.v2.train.load_checkpoint`
- `tf.train.load_checkpoint`

```
tf.train.load_checkpoint(ckpt_dir_or_file)

```

Defined in `python/training/checkpoint_utils.py`.
If `ckpt_dir_or_file` resolves to a directory with multiple checkpoints, reader for the latest checkpoint is returned.

Args:

- **`ckpt_dir_or_file`**: Directory with checkpoints file or path to checkpoint file.

Returns:
`CheckpointReader` object.

Raises:

- **`ValueError`**: If `ckpt_dir_or_file` resolves to a directory with no checkpoints.

tf.train.load_variable

- **Contents**

- Aliases:
Returns the tensor value of the given variable in the checkpoint.

Aliases:

- `tf.compat.v1.train.load_variable`
- `tf.compat.v2.train.load_variable`
- `tf.train.load_variable`

```
tf.train.load_variable(
    ckpt_dir_or_file,
    name
)
```

Defined in `python/training/checkpoint_utils.py`.

Args:

- **`ckpt_dir_or_file`**: Directory with checkpoints file or path to checkpoint.
- **`name`**: Name of the variable to return.

Returns:

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

tf.train.SequenceExample

- **Contents**
- Class SequenceExample
  - Aliases:
- Properties
  - context
  - feature_lists

Class `SequenceExample`

Aliases:

- Class `tf.compat.v1.train.SequenceExample`
- Class `tf.compat.v2.train.SequenceExample`
- Class `tf.train.SequenceExample`

Defined in `core/example/example.proto`.

Properties

`context`

`Features context`

`feature_lists`

`FeatureLists feature_lists`

tf.train.ServerDef

- **Contents**
- Class ServerDef
  - Aliases:
- Properties
  - cluster

Class `ServerDef`

Aliases:

- Class `tf.compat.v1.train.ServerDef`
- Class `tf.compat.v2.train.ServerDef`
- Class `tf.train.ServerDef`

Defined in `core/protobuf/tensorflow_server.proto`.

Properties

`cluster`

`ClusterDef cluster`

`default_session_config`

`ConfigProto default_session_config`

`job_name`

`string job_name`

`protocol`

`string protocol`

`task_index`

`int32 task_index`

Module: tf.train.experimental

- **Contents**
- Classes

Public API for tf.train.experimental namespace.

Classes

`class DynamicLossScale`: Loss scale that dynamically adjusts itself.
`class FixedLossScale`: Loss scale with a fixed value.
`class LossScale`: Loss scale base class.
`class PythonState`: A mixin for putting Python state in an object-based checkpoint.

tf.train.experimental.DynamicLossScale

- **Contents**
- Class DynamicLossScale
  - Aliases:
- __init__
- Properties

Class `DynamicLossScale`

Loss scale that dynamically adjusts itself.
Inherits From: `LossScale`

Aliases:

- Class `tf.compat.v1.train.experimental.DynamicLossScale`
- Class `tf.compat.v2.train.experimental.DynamicLossScale`
- Class `tf.train.experimental.DynamicLossScale`

Defined in `python/training/experimental/loss_scale.py`.

Dynamic loss scaling works by adjusting the loss scale as training progresses. The goal is to keep the loss scale as high as possible without overflowing the gradients. As long as the gradients do not overflow, raising the loss scale never hurts.
The algorithm starts by setting the loss scale to an initial value. Every N steps that the gradients are finite, the loss scale is increased by some factor. However, if a NaN or Inf gradient is found, the gradients for that step are not applied, and the loss scale is decreased by the factor. This process tends to keep the loss scale as high as possible without gradients overflowing.

`__init__`

```
__init__(
    initial_loss_scale=(2 ** 15),
    increment_period=2000,
    multiplier=2.0
)
```

Creates the dynamic loss scale.

Args:

- **`initial_loss_scale`**: A Python float. The loss scale to use at the beginning. It's better to start this at a very high number, because a loss scale that is too high gets lowered far more quickly than a loss scale that is to low gets raised. The default is 2 ** 15, which is approximately half the maximum float16 value.
- **`increment_period`**: Increases loss scale every `increment_period` consecutive steps that finite gradients are encountered. If a nonfinite gradient is encountered, the count is reset back to zero.
- **`multiplier`**: The multiplier to use when increasing or decreasing the loss scale.

Properties

`increment_period`

`initial_loss_scale`

`multiplier`

Methods

`__call__`

```
__call__()
```

`from_config`

```
from_config(
    cls,
    config
)
```

Creates the LossScale from its config.

`get_config`

```
get_config()
```

update

```
update(grads)
```

Updates loss scale based on if gradients are finite in current step.

tf.train.experimental.FixedLossScale

- **Contents**
- Class FixedLossScale
  - Aliases:
- __init__
- Methods

Class `FixedLossScale`

Loss scale with a fixed value.

Inherits From: `LossScale`

Aliases:

- Class `tf.compat.v1.train.experimental.FixedLossScale`
- Class `tf.compat.v2.train.experimental.FixedLossScale`
- Class `tf.train.experimental.FixedLossScale`

Defined in `python/training/experimental/loss_scale.py`.

The loss scale is not updated for the lifetime of instances of this class. A given instance of this class always returns the same number when called.

`__init__`

```
__init__(loss_scale_value)
```

Creates the fixed loss scale.

Args:

- **`loss_scale_value`**: A Python float. Its ideal value varies depending on models to run. Choosing a too small loss_scale might affect model quality; a too big loss_scale might cause inf or nan. There is no single right loss_scale to apply. There is no harm choosing a relatively big number as long as no nan or inf is encountered in training.

Raises:

- **`ValueError`**: If loss_scale is less than 1.

Methods

`__call__`

```
__call__()
```

`from_config`

```
from_config(
    cls,
    config
)
```

Creates the LossScale from its config.

get_config

```
get_config()
```

update

```
update(grads)
```

tf.train.experimental.LossScale

- **Contents**
- Class LossScale
  - Aliases:
- __init__
- Methods

Class `LossScale`

Loss scale base class.

Aliases:

- Class `tf.compat.v1.train.experimental.LossScale`
- Class `tf.compat.v2.train.experimental.LossScale`
- Class `tf.train.experimental.LossScale`

Defined in `python/training/experimental/loss_scale.py`.

Loss scaling is a process that multiplies the loss by a multiplier called the loss scale, and divides each gradient by the same multiplier. The pseudocode for this process is:

```
loss = ...
loss *= loss_scale
grads = gradients(loss, vars)
grads /= loss_scale
```

Mathematically, loss scaling has no effect, but can help avoid numerical underflow in intermediate gradients when float16 tensors are used for mixed precision training. By multiplying the loss, each intermediate gradient will have the same multiplier applied.

Instances of this class represent a loss scale. Calling instances of this class returns the loss scale as a scalar float32 tensor, while method `update()` updates the loss scale depending on the values of the gradients. Optimizers use instances of this class to scale loss and gradients.

__init__

```
__init__()
```

Initializes the loss scale class.

Methods

__call__

```
__call__()
```

Returns the current loss scale as a scalar `float32` tensor.

from_config

```
@classmethod
from_config(
    cls,
```

```
    config
)
```

Creates the LossScale from its config.

get_config

```
get_config()
```

Returns the config of this loss scale.

update

```
update(grads)
```

Updates the value of the loss scale.
The loss scale will be potentially updated, based on the value of `grads`. The tensor returned by calling this class is only updated when this function is evaluated.
In eager mode, this directly updates the loss scale, so that calling `__call__` will return the newly updated loss scale. In graph mode, this returns an op that, when evaluated, updates the loss scale.
This function also returns a `should_apply_gradients` bool. If False, gradients should not be applied to the variables that step, as nonfinite gradients were found, and the loss scale has been be updated to reduce the chance of finding nonfinite gradients in the next step. Some loss scale classes will always return True, as they cannot adjust themselves in response to nonfinite gradients.
When a DistributionStrategy is used, this function may only be called in a cross-replica context.

Args:

- **`grads`**: A list of unscaled gradients, each which is the gradient of the loss with respect to a weight. The gradients should have already been divided by the loss scale being before passed to this function. 'None' gradients are accepted, and are ignored.

Returns:

- **`update_op`**: In eager mode, None. In graph mode, an op to update the loss scale.
- **`should_apply_gradients`**: Either a bool or a scalar boolean tensor. If False, the caller should skip applying `grads` to the variables this step.

tf.train.experimental.PythonState

- **Contents**
- Class PythonState
  - Aliases:
- Methods
  - deserialize
  - serialize

Class `PythonState`

A mixin for putting Python state in an object-based checkpoint.

Aliases:

- Class `tf.compat.v1.train.experimental.PythonState`
- Class `tf.compat.v2.train.experimental.PythonState`
- Class `tf.train.experimental.PythonState`

Defined in `python/training/tracking/python_state.py`.
This is an abstract class which allows extensions to TensorFlow's object-based checkpointing (see `tf.train.Checkpoint`). For example a wrapper for NumPy arrays:

```
import io
import numpy

class NumpyWrapper(tf.train.experimental.PythonState):

  def __init__(self, array):
    self.array = array

  def serialize(self):
    string_file = io.BytesIO()
    try:
      numpy.save(string_file, self.array, allow_pickle=False)
      serialized = string_file.getvalue()
    finally:
      string_file.close()
    return serialized

  def deserialize(self, string_value):
    string_file = io.BytesIO(string_value)
    try:
      self.array = numpy.load(string_file, allow_pickle=False)
    finally:
      string_file.close()
```

Instances of `NumpyWrapper` are checkpointable objects, and will be saved and restored from checkpoints along with TensorFlow state like variables.

```
root = tf.train.Checkpoint(numpy=NumpyWrapper(numpy.array([1.])))
save_path = root.save(prefix)
root.numpy.array *= 2.
assert [2.] == root.numpy.array
root.restore(save_path)
assert [1.] == root.numpy.array
```

Methods

deserialize

```
deserialize(string_value)
```

Callback to deserialize the object.

serialize

```
serialize()
```

Callback to serialize the object. Returns a string.

Module: tf.compat.v1.feature_column

- **Contents**
- Functions

Public API for tf.feature_column namespace.

Functions

`bucketized_column(...)`: Represents discretized dense input.
`categorical_column_with_hash_bucket(...)`: Represents sparse feature where ids are set by hashing.
`categorical_column_with_identity(...)`: A `CategoricalColumn` that returns identity values.
`categorical_column_with_vocabulary_file(...)`: A `CategoricalColumn` with a vocabulary file.
`categorical_column_with_vocabulary_list(...)`: A `CategoricalColumn` with in-memory vocabulary.
`crossed_column(...)`: Returns a column for performing crosses of categorical features.
`embedding_column(...)`: `DenseColumn` that converts from sparse, categorical input.
`indicator_column(...)`: Represents multi-hot representation of given categorical column.
`input_layer(...)`: Returns a dense `Tensor` as input layer based on given `feature_columns`.
`linear_model(...)`: Returns a linear prediction `Tensor` based on given `feature_columns`.
`make_parse_example_spec(...)`: Creates parsing spec dictionary from input feature_columns.
`numeric_column(...)`: Represents real valued or numerical features.
`sequence_categorical_column_with_hash_bucket(...)`: A sequence of categorical terms where ids are set by hashing.
`sequence_categorical_column_with_identity(...)`: Returns a feature column that represents sequences of integers.
`sequence_categorical_column_with_vocabulary_file(...)`: A sequence of categorical terms where ids use a vocabulary file.
`sequence_categorical_column_with_vocabulary_list(...)`: A sequence of categorical terms where ids use an in-memory list.
`sequence_numeric_column(...)`: Returns a feature column that represents sequences of numeric data.
`shared_embedding_columns(...)`: List of dense columns that convert from sparse, categorical input.
`weighted_categorical_column(...)`: Applies weight values to a `CategoricalColumn`.

tf.compat.v1.feature_column.input_layer

Returns a dense `Tensor` as input layer based on given `feature_columns`.

```
tf.compat.v1.feature_column.input_layer(

    features,

    feature_columns,

    weight_collections=None,

    trainable=True,

    cols_to_vars=None,

    cols_to_output_tensors=None

)
```

Defined in `python/feature_column/feature_column.py`.

Generally a single example in training data is described with FeatureColumns. At the first layer of the model, this column oriented data should be converted to a single `Tensor`.

Example:

```
price = numeric_column('price')

keywords_embedded = embedding_column(

    categorical_column_with_hash_bucket("keywords", 10K), dimensions=16)

columns = [price, keywords_embedded, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)

for units in [128, 64, 32]:

  dense_tensor = tf.compat.v1.layers.dense(dense_tensor, units, tf.nn.relu)

prediction = tf.compat.v1.layers.dense(dense_tensor, 1)
```

Args:

- `features`: A mapping from key to tensors. `_FeatureColumn`s look up via these keys. For example `numeric_column('price')` will look at 'price' key in this dict. Values can be a `SparseTensor` or a `Tensor` depends on corresponding `_FeatureColumn`.
- `feature_columns`: An iterable containing the FeatureColumns to use as inputs to your model. All items should be instances of classes derived from `_DenseColumn` such as `numeric_column`, `embedding_column`, `bucketized_column`, `indicator_column`. If you have categorical features, you can wrap them with an `embedding_column` or `indicator_column`.
- `weight_collections`: A list of collection names to which the Variable will be added. Note that variables will also be added to collections `tf.GraphKeys.GLOBAL_VARIABLES` and `ops.GraphKeys.MODEL_VARIABLES`.
- `trainable`: If `True` also add the variable to the graph collection`GraphKeys.TRAINABLE_VARIABLES` (see `tf.Variable`).
- `cols_to_vars`: If not `None`, must be a dictionary that will be filled with a mapping from `_FeatureColumn` to list of `Variable`s. For example, after the call, we might have cols_to_vars = {_EmbeddingColumn(categorical_column=_HashedCategoricalColumn(key='sparse_feature', hash_bucket_size=5, dtype=tf.string), dimension=10): [<tf.Variable 'some_variable:0' shape=(5, 10), <tf.Variable 'some_variable:1' shape=(5, 10)]} If a column creates no variables, its value will be an empty list.
- `cols_to_output_tensors`: If not `None`, must be a dictionary that will be filled with a mapping from '_FeatureColumn' to the associated output `Tensor`s.

Returns:

A `Tensor` which represents input layer of a model. Its shape is (batch_size, first_layer_dimension) and its dtype is `float32`. first_layer_dimension is determined based on given `feature_columns`.

Raises:

- `ValueError`: if an item in `feature_columns` is not a `_DenseColumn`.

tf.compat.v1.feature_column.linear_model

Returns a linear prediction `Tensor` based on given `feature_columns`.

```
tf.compat.v1.feature_column.linear_model(

    features,

    feature_columns,

    units=1,

    sparse_combiner='sum',

    weight_collections=None,

    trainable=True,

    cols_to_vars=None

)
```

Defined in `python/feature_column/feature_column.py`.

This function generates a weighted sum based on output dimension `units`. Weighted sum refers to logits in classification problems. It refers to the prediction itself for linear regression problems.

Note on supported columns: `linear_model` treats categorical columns as `indicator_column`s. To be specific, assume the input as `SparseTensor` looks like:

```
  shape = [2, 2]

  {

      [0, 0]: "a"

      [1, 0]: "b"

      [1, 1]: "c"

  }
```

`linear_model` assigns weights for the presence of "a", "b", "c' implicitly, just like `indicator_column`, while `input_layer` explicitly requires wrapping each of categorical columns with an `embedding_column` or an `indicator_column`.

Example of usage:

```
price = numeric_column('price')

price_buckets = bucketized_column(price, boundaries=[0., 10., 100., 1000.])
```

```
keywords = categorical_column_with_hash_bucket("keywords", 10K)

keywords_price = crossed_column('keywords', price_buckets, ...)

columns = [price_buckets, keywords, keywords_price ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

prediction = linear_model(features, columns)
```

The `sparse_combiner` argument works as follows For example, for two features represented as the categorical columns:

```
  # Feature 1


  shape = [2, 2]

  {

      [0, 0]: "a"

      [0, 1]: "b"

      [1, 0]: "c"

  }



  # Feature 2


  shape = [2, 3]

  {

      [0, 0]: "d"

      [1, 0]: "e"

      [1, 1]: "f"

      [1, 2]: "f"
```

```
}
```

with `sparse_combiner` as "mean", the linear model outputs consequently are:

```
y_0 = 1.0 / 2.0 * ( w_a + w_b ) + w_d + b

y_1 = w_c + 1.0 / 3.0 * ( w_e + 2.0 * w_f ) + b
```

where `y_i` is the output, `b` is the bias, and `w_x` is the weight assigned to the presence of `x` in the input features.

Args:

- `features`: A mapping from key to tensors. `_FeatureColumn`s look up via these keys. For example `numeric_column('price')` will look at 'price' key in this dict. Values are `Tensor` or `SparseTensor` depending on corresponding `_FeatureColumn`.
- `feature_columns`: An iterable containing the FeatureColumns to use as inputs to your model. All items should be instances of classes derived from `_FeatureColumn`s.
- `units`: An integer, dimensionality of the output space. Default value is 1.
- `sparse_combiner`: A string specifying how to reduce if a categorical column is multivalent. Except `numeric_column`, almost all columns passed to `linear_model` are considered as categorical columns. It combines each categorical column independently. Currently "mean", "sqrtn" and "sum" are supported, with "sum" the default for linear model. "sqrtn" often achieves good accuracy, in particular with bag-of-words columns.
- "sum": do not normalize features in the column
- "mean": do l1 normalization on features in the column
- "sqrtn": do l2 normalization on features in the column
- `weight_collections`: A list of collection names to which the Variable will be added. Note that, variables will also be added to collections `tf.GraphKeys.GLOBAL_VARIABLES` and `ops.GraphKeys.MODEL_VARIABLES`.
- `trainable`: If `True` also add the variable to the graph collection`GraphKeys.TRAINABLE_VARIABLES` (see `tf.Variable`).
- `cols_to_vars`: If not `None`, must be a dictionary that will be filled with a mapping from `_FeatureColumn` to associated list of `Variable`s. For example, after the call, we might have cols_to_vars = { _NumericColumn(key='numeric_feature1', shape=(1,): [], 'bias': [], _NumericColumn(key='numeric_feature2', shape=(2,)): []} If a column creates no variables, its value will be an empty list. Note that cols_to_vars will also contain a string key 'bias' that maps to a list of Variables.

Returns:

A `Tensor` which represents predictions/logits of a linear model. Its shape is (batch_size, units) and its dtype is `float32`.

Raises:

- `ValueError`: if an item in `feature_columns` is neither a `_DenseColumn` nor `_CategoricalColumn`.

tf.compat.v1.feature_column.shared_embedding_columns

List of dense columns that convert from sparse, categorical input.

```
tf.compat.v1.feature_column.shared_embedding_columns(
```

```
    categorical_columns,

    dimension,

    combiner='mean',

    initializer=None,

    shared_embedding_collection_name=None,

    ckpt_to_load_from=None,

    tensor_name_in_ckpt=None,

    max_norm=None,

    trainable=True

)
```

Defined in `python/feature_column/feature_column_v2.py`.
This is similar to `embedding_column`, except that it produces a list of embedding columns that share the same embedding weights.
Use this when your inputs are sparse and of the same type (e.g. watched and impression video IDs that share the same vocabulary), and you want to convert them to a dense representation (e.g., to feed to a DNN).
Inputs must be a list of categorical columns created by any of the `categorical_column_*` function. They must all be of the same type and have the same arguments except `key`. E.g. they can be categorical_column_with_vocabulary_file with the same vocabulary_file. Some or all columns could also be weighted_categorical_column.
Here is an example embedding of two features for a DNNClassifier model:

```
watched_video_id = categorical_column_with_vocabulary_file(

    'watched_video_id', video_vocabulary_file, video_vocabulary_size)

impression_video_id = categorical_column_with_vocabulary_file(

    'impression_video_id', video_vocabulary_file, video_vocabulary_size)

columns = shared_embedding_columns(

    [watched_video_id, impression_video_id], dimension=10)



estimator = tf.estimator.DNNClassifier(feature_columns=columns, ...)
```

```
label_column = ...
def input_fn():
  features = tf.io.parse_example(
      ..., features=make_parse_example_spec(columns + [label_column]))
  labels = features.pop(label_column.name)
  return features, labels


estimator.train(input_fn=input_fn, steps=100)
```

Here is an example using `shared_embedding_columns` with model_fn:

```
def model_fn(features, ...):
  watched_video_id = categorical_column_with_vocabulary_file(
      'watched_video_id', video_vocabulary_file, video_vocabulary_size)
  impression_video_id = categorical_column_with_vocabulary_file(
      'impression_video_id', video_vocabulary_file, video_vocabulary_size)
  columns = shared_embedding_columns(
      [watched_video_id, impression_video_id], dimension=10)
  dense_tensor = input_layer(features, columns)
  # Form DNN layers, calculate loss, and return EstimatorSpec.
  ...
```

Args:

- `categorical_columns`: List of categorical columns created by a`categorical_column_with_*` function. These columns produce the sparse IDs that are inputs to the embedding lookup. All columns must be of the same type and have the same arguments except `key`. E.g. they can be categorical_column_with_vocabulary_file with the same vocabulary_file. Some or all columns could also be weighted_categorical_column.
- `dimension`: An integer specifying dimension of the embedding, must be > 0.

- `combiner`: A string specifying how to reduce if there are multiple entries in a single row. Currently 'mean', 'sqrtn' and 'sum' are supported, with 'mean' the default. 'sqrtn' often achieves good accuracy, in particular with bag-of-words columns. Each of this can be thought as example level normalizations on the column. For more information, see `tf.embedding_lookup_sparse`.
- `initializer`: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to `tf.compat.v1.truncated_normal_initializer` with mean `0.0` and standard deviation `1/sqrt(dimension)`.
- `shared_embedding_collection_name`: Optional name of the collection where shared embedding weights are added. If not given, a reasonable name will be chosen based on the names of `categorical_columns`. This is also used in `variable_scope` when creating shared embedding weights.
- `ckpt_to_load_from`: String representing checkpoint name/pattern from which to restore column weights. Required if `tensor_name_in_ckpt` is not `None`.
- `tensor_name_in_ckpt`: Name of the `Tensor` in `ckpt_to_load_from` from which to restore the column weights. Required if `ckpt_to_load_from` is not `None`.
- `max_norm`: If not `None`, each embedding is clipped if its l2-norm is larger than this value, before combining.
- `trainable`: Whether or not the embedding is trainable. Default is True.

Returns:
A list of dense columns that converts from sparse input. The order of results follows the ordering of `categorical_columns`.

Raises:
- `ValueError`: if `dimension` not > 0.
- `ValueError`: if any of the given `categorical_columns` is of different type or has different arguments than the others.
- `ValueError`: if exactly one of `ckpt_to_load_from` and `tensor_name_in_ckpt` is specified.
- `ValueError`: if `initializer` is specified and is not callable.
- `RuntimeError`: if eager execution is enabled.

tf.feature_column.bucketized_column

- **Contents**
- Aliases:
- Used in the tutorials:

Represents discretized dense input.

Aliases:
- `tf.compat.v1.feature_column.bucketized_column`
- `tf.compat.v2.feature_column.bucketized_column`
- `tf.feature_column.bucketized_column`

```
tf.feature_column.bucketized_column(

    source_column,

    boundaries

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:

- [Classify structured data]()

Buckets include the left boundary, and exclude the right boundary. Namely, `boundaries=[0., 1., 2.]` generates buckets `(-inf, 0.)`, `[0., 1.)`, `[1., 2.)`, and `[2., +inf)`.
For example, if the inputs are

```
boundaries = [0, 10, 100]

input tensor = [[-5, 10000]

                [150,   10]

                [5,    100]]
```

then the output will be

```
output = [[0, 3]

          [3, 2]

          [1, 3]]
```

Example:

```
price = numeric_column('price')

bucketized_price = bucketized_column(price, boundaries=[...])

columns = [bucketized_price, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)



# or

columns = [bucketized_price, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)
```

`bucketized_column` can also be crossed with another categorical column using `crossed_column`:

```
price = numeric_column('price')
```

```
# bucketized_column converts numerical feature to a categorical one.

bucketized_price = bucketized_column(price, boundaries=[...])

# 'keywords' is a string feature.

price_x_keywords = crossed_column([bucketized_price, 'keywords'], 50K)

columns = [price_x_keywords, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

Args:

- `source_column`: A one-dimensional dense column which is generated with `numeric_column`.
- `boundaries`: A sorted list or tuple of floats specifying the boundaries.

Returns:

A `BucketizedColumn`.

Raises:

- `ValueError`: If `source_column` is not a numeric column, or if it is not one-dimensional.
- `ValueError`: If `boundaries` is not a sorted list or tuple.

tf.feature_column.categorical_column_with_hash_bucket

- **Contents**
- Aliases:
- Used in the tutorials:

Represents sparse feature where ids are set by hashing.

Aliases:

- `tf.compat.v1.feature_column.categorical_column_with_hash_bucket`
- `tf.compat.v2.feature_column.categorical_column_with_hash_bucket`
- `tf.feature_column.categorical_column_with_hash_bucket`

```
tf.feature_column.categorical_column_with_hash_bucket(

    key,

    hash_bucket_size,

    dtype=tf.dtypes.string

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:

- Classify structured data

Use this when your sparse features are in string or integer format, and you want to distribute your inputs into a finite number of buckets by hashing. output_id = Hash(input_feature_string) % bucket_size for string type input. For int type input, the value is converted to its string representation first and then hashed by the same formula.

For input dictionary `features`, `features[key]` is either `Tensor` or `SparseTensor`. If `Tensor`, missing values can be represented by `-1` for int and `''` for string, which will be dropped by this feature column.

Example:

```
keywords = categorical_column_with_hash_bucket("keywords", 10K)

columns = [keywords, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)



# or

keywords_embedded = embedding_column(keywords, 16)

columns = [keywords_embedded, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)
```

Args:

- `key`: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature `Tensor` objects, and feature columns.
- `hash_bucket_size`: An int > 1. The number of buckets.
- `dtype`: The type of features. Only string and integer types are supported.

Returns:

A `HashedCategoricalColumn`.

Raises:

- `ValueError`: `hash_bucket_size` is not greater than 1.
- `ValueError`: `dtype` is neither string nor integer.

tf.feature_column.categorical_column_with_identity

- **Contents**
- Aliases:

A `CategoricalColumn` that returns identity values.

Aliases:

- `tf.compat.v1.feature_column.categorical_column_with_identity`
- `tf.compat.v2.feature_column.categorical_column_with_identity`
- `tf.feature_column.categorical_column_with_identity`

```
tf.feature_column.categorical_column_with_identity(

    key,

    num_buckets,

    default_value=None

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Use this when your inputs are integers in the range `[0, num_buckets)`, and you want to use the input value itself as the categorical ID. Values outside this range will result in `default_value` if specified, otherwise it will fail.

Typically, this is used for contiguous ranges of integer indexes, but it doesn't have to be. This might be inefficient, however, if many of IDs are unused.

Consider `categorical_column_with_hash_bucket` in that case.

For input dictionary `features`, `features[key]` is either `Tensor` or `SparseTensor`. If `Tensor`, missing values can be represented by `-1` for int and `''` for string, which will be dropped by this feature column.

In the following examples, each input in the range `[0, 1000000)` is assigned the same value. All other inputs are assigned `default_value` 0. Note that a literal 0 in inputs will result in the same default ID.

Linear model:

```
video_id = categorical_column_with_identity(

    key='video_id', num_buckets=1000000, default_value=0)

columns = [video_id, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction, _, _ = linear_model(features, columns)
```

Embedding for a DNN model:

```
columns = [embedding_column(video_id, 9),...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)
```

Args:

- `key`: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature `Tensor` objects, and feature columns.
- `num_buckets`: Range of inputs and outputs is `[0, num_buckets)`.
- `default_value`: If `None`, this column's graph operations will fail for out-of-range inputs. Otherwise, this value must be in the range `[0, num_buckets)`, and will replace inputs in that range.

Returns:

A `CategoricalColumn` that returns identity values.

Raises:

- `ValueError`: if `num_buckets` is less than one.
- `ValueError`: if `default_value` is not in range `[0, num_buckets)`.

tf.feature_column.categorical_column_with_vocabulary_file

- **Contents**
- Aliases:

A `CategoricalColumn` with a vocabulary file.

Aliases:

- `tf.compat.v2.feature_column.categorical_column_with_vocabulary_file`
- `tf.feature_column.categorical_column_with_vocabulary_file`

```
tf.feature_column.categorical_column_with_vocabulary_file(

    key,

    vocabulary_file,

    vocabulary_size=None,

    dtype=tf.dtypes.string,

    default_value=None,

    num_oov_buckets=0

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Use this when your inputs are in string or integer format, and you have a vocabulary file that maps each value to an integer ID. By default, out-of-vocabulary values are ignored. Use either (but not both) of`num_oov_buckets` and `default_value` to specify how to include out-of-vocabulary values.

For input dictionary `features`, `features[key]` is either `Tensor` or `SparseTensor`. If `Tensor`, missing values can be represented by `-1` for int and `''` for string, which will be dropped by this feature column.

Example with `num_oov_buckets`: File `'/us/states.txt'` contains 50 lines, each with a 2-character U.S. state abbreviation. All inputs with values in that file are assigned an ID 0-49, corresponding to its line number. All other values are hashed and assigned an ID 50-54.

```
states = categorical_column_with_vocabulary_file(

    key='states', vocabulary_file='/us/states.txt', vocabulary_size=50,

    num_oov_buckets=5)

columns = [states, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

Example with `default_value`: File `'/us/states.txt'` contains 51 lines - the first line is `'XX'`, and the other 50 each have a 2-character U.S. state abbreviation. Both a literal `'XX'` in input, and other values missing from the file, will be assigned ID 0. All others are assigned the corresponding line number 1-50.

```
states = categorical_column_with_vocabulary_file(

    key='states', vocabulary_file='/us/states.txt', vocabulary_size=51,

    default_value=0)

columns = [states, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction, _, _ = linear_model(features, columns)
```

And to make an embedding with either:

```
columns = [embedding_column(states, 3),...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)
```

Args:

- `key`: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature `Tensor` objects, and feature columns.
- `vocabulary_file`: The vocabulary file name.
- `vocabulary_size`: Number of the elements in the vocabulary. This must be no greater than length of `vocabulary_file`, if less than length, later values are ignored. If None, it is set to the length of `vocabulary_file`.
- `dtype`: The type of features. Only string and integer types are supported.
- `default_value`: The integer ID value to return for out-of-vocabulary feature values, defaults to `-1`. This can not be specified with a positive `num_oov_buckets`.

- `num_oov_buckets`: Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range `[vocabulary_size, vocabulary_size+num_oov_buckets)` based on a hash of the input value. A positive `num_oov_buckets` can not be specified with `default_value`.

Returns:
A `CategoricalColumn` with a vocabulary file.

Raises:
- `ValueError`: `vocabulary_file` is missing or cannot be opened.
- `ValueError`: `vocabulary_size` is missing or < 1.
- `ValueError`: `num_oov_buckets` is a negative integer.
- `ValueError`: `num_oov_buckets` and `default_value` are both specified.
- `ValueError`: `dtype` is neither string nor integer.

tf.feature_column.categorical_column_with_vocabulary_list

- **Contents**
- Aliases:
- Used in the tutorials:

A `CategoricalColumn` with in-memory vocabulary.

Aliases:
- `tf.compat.v1.feature_column.categorical_column_with_vocabulary_list`
- `tf.compat.v2.feature_column.categorical_column_with_vocabulary_list`
- `tf.feature_column.categorical_column_with_vocabulary_list`

```
tf.feature_column.categorical_column_with_vocabulary_list(

    key,

    vocabulary_list,

    dtype=None,

    default_value=-1,

    num_oov_buckets=0

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:
- Build a linear model with Estimators
- Classify structured data

Use this when your inputs are in string or integer format, and you have an in-memory vocabulary mapping each value to an integer ID. By default, out-of-vocabulary values are ignored. Use either (but not both) of `num_oov_buckets` and `default_value` to specify how to include out-of-vocabulary values.

For input dictionary `features`, `features[key]` is either `Tensor` or `SparseTensor`. If `Tensor`, missing values can be represented by `-1` for int and `''` for string, which will be dropped by this feature column.
Example with `num_oov_buckets`: In the following example, each input in `vocabulary_list` is assigned an ID 0-3 corresponding to its index (e.g., input 'B' produces output 2). All other inputs are hashed and assigned an ID 4-5.

```
colors = categorical_column_with_vocabulary_list(

    key='colors', vocabulary_list=('R', 'G', 'B', 'Y'),

    num_oov_buckets=2)

columns = [colors, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction, _, _ = linear_model(features, columns)
```

Example with `default_value`: In the following example, each input in `vocabulary_list` is assigned an ID 0-4 corresponding to its index (e.g., input 'B' produces output 3). All other inputs are assigned `default_value` 0.

```
colors = categorical_column_with_vocabulary_list(

    key='colors', vocabulary_list=('X', 'R', 'G', 'B', 'Y'), default_value=0)

columns = [colors, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction, _, _ = linear_model(features, columns)
```

And to make an embedding with either:

```
columns = [embedding_column(colors, 3),...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)
```

Args:

- `key`: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature `Tensor` objects, and feature columns.
- `vocabulary_list`: An ordered iterable defining the vocabulary. Each feature is mapped to the index of its value (if present) in `vocabulary_list`. Must be castable to `dtype`.
- `dtype`: The type of features. Only string and integer types are supported. If `None`, it will be inferred from `vocabulary_list`.

- `default_value`: The integer ID value to return for out-of-vocabulary feature values, defaults to `-1`. This can not be specified with a positive `num_oov_buckets`.
- `num_oov_buckets`: Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range `[len(vocabulary_list), len(vocabulary_list)+num_oov_buckets)` based on a hash of the input value. A positive `num_oov_buckets` can not be specified with `default_value`.

Returns:
A `CategoricalColumn` with in-memory vocabulary.

Raises:
- `ValueError`: if `vocabulary_list` is empty, or contains duplicate keys.
- `ValueError`: `num_oov_buckets` is a negative integer.
- `ValueError`: `num_oov_buckets` and `default_value` are both specified.
- `ValueError`: if `dtype` is not integer or string.

tf.feature_column.crossed_column

- **Contents**
- Aliases:
- Used in the tutorials:

Returns a column for performing crosses of categorical features.

Aliases:
- `tf.compat.v1.feature_column.crossed_column`
- `tf.compat.v2.feature_column.crossed_column`
- `tf.feature_column.crossed_column`

```
tf.feature_column.crossed_column(
    keys,
    hash_bucket_size,
    hash_key=None
)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:
- Build a linear model with Estimators
- Classify structured data

Crossed features will be hashed according to `hash_bucket_size`. Conceptually, the transformation can be thought of as: Hash(cartesian product of features) % `hash_bucket_size`
For example, if the input features are:
- SparseTensor referred by first key:

```
shape = [2, 2]
{
    [0, 0]: "a"
```

```
    [1, 0]: "b"

    [1, 1]: "c"

}
```

- SparseTensor referred by second key:

```
shape = [2, 1]
```

```
{

    [0, 0]: "d"

    [1, 0]: "e"

}
```

then crossed feature will look like:

```
 shape = [2, 2]

{

    [0, 0]: Hash64("d", Hash64("a")) % hash_bucket_size

    [1, 0]: Hash64("e", Hash64("b")) % hash_bucket_size

    [1, 1]: Hash64("e", Hash64("c")) % hash_bucket_size

}
```

Here is an example to create a linear model with crosses of string features:

```
keywords_x_doc_terms = crossed_column(['keywords', 'doc_terms'], 50K)

columns = [keywords_x_doc_terms, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

You could also use vocabulary lookup before crossing:

```
keywords = categorical_column_with_vocabulary_file(

    'keywords', '/path/to/vocabulary/file', vocabulary_size=1K)
```

```
keywords_x_doc_terms = crossed_column([keywords, 'doc_terms'], 50K)

columns = [keywords_x_doc_terms, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

If an input feature is of numeric type, you can use `categorical_column_with_identity`, or `bucketized_column`, as in the example:

```
# vertical_id is an integer categorical feature.

vertical_id = categorical_column_with_identity('vertical_id', 10K)

price = numeric_column('price')

# bucketized_column converts numerical feature to a categorical one.

bucketized_price = bucketized_column(price, boundaries=[...])

vertical_id_x_price = crossed_column([vertical_id, bucketized_price], 50K)

columns = [vertical_id_x_price, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

To use crossed column in DNN model, you need to add it in an embedding column as in this example:

```
vertical_id_x_price = crossed_column([vertical_id, bucketized_price], 50K)

vertical_id_x_price_embedded = embedding_column(vertical_id_x_price, 10)

dense_tensor = input_layer(features, [vertical_id_x_price_embedded, ...])
```

Args:

- `keys`: An iterable identifying the features to be crossed. Each element can be either:
- string: Will use the corresponding feature which must be of string type.
- `CategoricalColumn`: Will use the transformed tensor produced by this column. Does not support hashed categorical column.
- `hash_bucket_size`: An int > 1. The number of buckets.
- `hash_key`: Specify the hash_key that will be used by the `FingerprintCat64` function to combine the crosses fingerprints on SparseCrossOp (optional).

Returns:
A `CrossedColumn`.

Raises:

- `ValueError`: If `len(keys) < 2`.
- `ValueError`: If any of the keys is neither a string nor `CategoricalColumn`.
- `ValueError`: If any of the keys is `HashedCategoricalColumn`.
- `ValueError`: If `hash_bucket_size < 1`.

tf.feature_column.embedding_column

- **Contents**
- Aliases:
- Used in the tutorials:

`DenseColumn` that converts from sparse, categorical input.

Aliases:

- `tf.compat.v1.feature_column.embedding_column`
- `tf.compat.v2.feature_column.embedding_column`
- `tf.feature_column.embedding_column`

```
tf.feature_column.embedding_column(

    categorical_column,

    dimension,

    combiner='mean',

    initializer=None,

    ckpt_to_load_from=None,

    tensor_name_in_ckpt=None,

    max_norm=None,

    trainable=True

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:

- Classify structured data

Use this when your inputs are sparse, but you want to convert them to a dense representation (e.g., to feed to a DNN).
Inputs must be a `CategoricalColumn` created by any of the `categorical_column_*` function. Here is an example of using `embedding_column` with `DNNClassifier`:

```
video_id = categorical_column_with_identity(
```

```
    key='video_id', num_buckets=1000000, default_value=0)
columns = [embedding_column(video_id, 9),...]


estimator = tf.estimator.DNNClassifier(feature_columns=columns, ...)


label_column = ...
def input_fn():
  features = tf.io.parse_example(
      ..., features=make_parse_example_spec(columns + [label_column]))
  labels = features.pop(label_column.name)
  return features, labels


estimator.train(input_fn=input_fn, steps=100)
```

Here is an example using `embedding_column` with model_fn:

```
def model_fn(features, ...):
  video_id = categorical_column_with_identity(
      key='video_id', num_buckets=1000000, default_value=0)
  columns = [embedding_column(video_id, 9),...]
  dense_tensor = input_layer(features, columns)
  # Form DNN layers, calculate loss, and return EstimatorSpec.
  ...
```

Args:

- `categorical_column`: A `CategoricalColumn` created by a `categorical_column_with_*`function. This column produces the sparse IDs that are inputs to the embedding lookup.
- `dimension`: An integer specifying dimension of the embedding, must be > 0.

- `combiner`: A string specifying how to reduce if there are multiple entries in a single row. Currently 'mean', 'sqrtn' and 'sum' are supported, with 'mean' the default. 'sqrtn' often achieves good accuracy, in particular with bag-of-words columns. Each of this can be thought as example level normalizations on the column. For more information, see `tf.embedding_lookup_sparse`.
- `initializer`: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to `tf.compat.v1.truncated_normal_initializer` with mean `0.0` and standard deviation `1/sqrt(dimension)`.
- `ckpt_to_load_from`: String representing checkpoint name/pattern from which to restore column weights. Required if `tensor_name_in_ckpt` is not `None`.
- `tensor_name_in_ckpt`: Name of the `Tensor` in `ckpt_to_load_from` from which to restore the column weights. Required if `ckpt_to_load_from` is not `None`.
- `max_norm`: If not `None`, embedding values are l2-normalized to this value.
- `trainable`: Whether or not the embedding is trainable. Default is True.

Returns:

`DenseColumn` that converts from sparse input.

Raises:

- `ValueError`: if `dimension` not > 0.
- `ValueError`: if exactly one of `ckpt_to_load_from` and `tensor_name_in_ckpt` is specified.
- `ValueError`: if `initializer` is specified and is not callable.
- `RuntimeError`: If eager execution is enabled.

tf.feature_column.indicator_column

- **Contents**
- Aliases:
- Used in the tutorials:

Represents multi-hot representation of given categorical column.

Aliases:

- `tf.compat.v1.feature_column.indicator_column`
- `tf.compat.v2.feature_column.indicator_column`
- `tf.feature_column.indicator_column`

```
tf.feature_column.indicator_column(categorical_column)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the tutorials:

- Build a linear model with Estimators
- Classify structured data
- For DNN model, `indicator_column` can be used to wrap any `categorical_column_*` (e.g., to feed to DNN). Consider to Use `embedding_column` if the number of buckets/unique(values) are large.
- For Wide (aka linear) model, `indicator_column` is the internal representation for categorical column when passing categorical column directly (as any element in feature_columns) to `linear_model`. See `linear_model` for details.

```
name = indicator_column(categorical_column_with_vocabulary_list(

    'name', ['bob', 'george', 'wanda'])

columns = [name, ...]
```

```
features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)



dense_tensor == [[1, 0, 0]]  # If "name" bytes_list is ["bob"]

dense_tensor == [[1, 0, 1]]  # If "name" bytes_list is ["bob", "wanda"]

dense_tensor == [[2, 0, 0]]  # If "name" bytes_list is ["bob", "bob"]
```

Args:

- `categorical_column`: A `CategoricalColumn` which is created by`categorical_column_with_*` or `crossed_column` functions.

Returns:
An `IndicatorColumn`.

tf.feature_column.make_parse_example_spec

- **Contents**
- Aliases:
- Used in the guide:

Creates parsing spec dictionary from input feature_columns.

Aliases:

- `tf.compat.v2.feature_column.make_parse_example_spec`
- `tf.feature_column.make_parse_example_spec`

```
tf.feature_column.make_parse_example_spec(feature_columns)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the guide:

- [Using the SavedModel format]

The returned dictionary can be used as arg 'features' in `tf.io.parse_example`.

Typical usage example:

```
# Define features and transformations

feature_a = categorical_column_with_vocabulary_file(...)

feature_b = numeric_column(...)

feature_c_bucketized = bucketized_column(numeric_column("feature_c"), ...)

feature_a_x_feature_c = crossed_column(

    columns=["feature_a", feature_c_bucketized], ...)
```

```
feature_columns = set(
    [feature_b, feature_c_bucketized, feature_a_x_feature_c])
features = tf.io.parse_example(
    serialized=serialized_examples,
    features=make_parse_example_spec(feature_columns))
```

For the above example, make_parse_example_spec would return the dict:

```
{
    "feature_a": parsing_ops.VarLenFeature(tf.string),
    "feature_b": parsing_ops.FixedLenFeature([1], dtype=tf.float32),
    "feature_c": parsing_ops.FixedLenFeature([1], dtype=tf.float32)
}
```

Args:

- `feature_columns`: An iterable containing all feature columns. All items should be instances of classes derived from `FeatureColumn`.

Returns:

A dict mapping each feature key to a `FixedLenFeature` or `VarLenFeature` value.

Raises:

- `ValueError`: If any of the given `feature_columns` is not a `FeatureColumn` instance.

tf.feature_column.numeric_column

- **Contents**
- Aliases:
- Used in the guide:
- Used in the tutorials:

Represents real valued or numerical features.

Aliases:

- `tf.compat.v1.feature_column.numeric_column`
- `tf.compat.v2.feature_column.numeric_column`
- `tf.feature_column.numeric_column`

```
tf.feature_column.numeric_column(
    key,
```

```
    shape=(1,),

    default_value=None,

    dtype=tf.dtypes.float32,

    normalizer_fn=None

)
```

Defined in `python/feature_column/feature_column_v2.py`.

Used in the guide:

- Distributed training in TensorFlow
- Using the SavedModel format

Used in the tutorials:

- Build a linear model with Estimators
- Classify structured data
- Premade Estimators

Example:

```
price = numeric_column('price')

columns = [price, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

dense_tensor = input_layer(features, columns)



# or

bucketized_price = bucketized_column(price, boundaries=[...])

columns = [bucketized_price, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction = linear_model(features, columns)
```

Args:

- `key`: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature `Tensor` objects, and feature columns.
- `shape`: An iterable of integers specifies the shape of the `Tensor`. An integer can be given which means a single dimension `Tensor` with given width. The `Tensor` representing the column will have the shape of [batch_size] + `shape`.

- `default_value`: A single value compatible with `dtype` or an iterable of values compatible with `dtype` which the column takes on during `tf.Example` parsing if data is missing. A default value of `None` will cause `tf.io.parse_example` to fail if an example does not contain this column. If a single value is provided, the same value will be applied as the default value for every item. If an iterable of values is provided, the shape of the `default_value` should be equal to the given `shape`.
- `dtype`: defines the type of values. Default value is `tf.float32`. Must be a non-quantized, real integer or floating point type.
- `normalizer_fn`: If not `None`, a function that can be used to normalize the value of the tensor after `default_value` is applied for parsing. Normalizer function takes the input `Tensor` as its argument, and returns the output `Tensor`. (e.g. lambda x: (x - 3.0) / 4.2). Please note that even though the most common use case of this function is normalization, it can be used for any kind of Tensorflow transformations.

Returns:

A `NumericColumn`.

Raises:

- `TypeError`: if any dimension in shape is not an int
- `ValueError`: if any dimension in shape is not a positive integer
- `TypeError`: if `default_value` is an iterable but not compatible with `shape`
- `TypeError`: if `default_value` is not compatible with `dtype`.
- `ValueError`: if `dtype` is not convertible to `tf.float32`.

tf.feature_column.sequence_categorical_column_with_hash_bucket

- **Contents**
- Aliases:

A sequence of categorical terms where ids are set by hashing.

Aliases:

- `tf.compat.v1.feature_column.sequence_categorical_column_with_hash_bucket`
- `tf.compat.v2.feature_column.sequence_categorical_column_with_hash_bucket`
- `tf.feature_column.sequence_categorical_column_with_hash_bucket`

```
tf.feature_column.sequence_categorical_column_with_hash_bucket(

    key,

    hash_bucket_size,

    dtype=tf.dtypes.string

)
```

Defined in `python/feature_column/sequence_feature_column.py`.

Pass this to `embedding_column` or `indicator_column` to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

Example:

```
tokens = sequence_categorical_column_with_hash_bucket(
```

```
    'tokens', hash_bucket_size=1000)

tokens_embedding = embedding_column(tokens, dimension=10)

columns = [tokens_embedding]



features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

sequence_feature_layer = SequenceFeatures(columns)

sequence_input, sequence_length = sequence_feature_layer(features)

sequence_length_mask = tf.sequence_mask(sequence_length)



rnn_cell = tf.keras.layers.SimpleRNNCell(hidden_size)

rnn_layer = tf.keras.layers.RNN(rnn_cell)

outputs, state = rnn_layer(sequence_input, mask=sequence_length_mask)
```

Args:

- `key`: A unique string identifying the input feature.
- `hash_bucket_size`: An int > 1. The number of buckets.
- `dtype`: The type of features. Only string and integer types are supported.

Returns:

A `SequenceCategoricalColumn`.

Raises:

- `ValueError`: `hash_bucket_size` is not greater than 1.
- `ValueError`: `dtype` is neither string nor integer.

tf.feature_column.sequence_categorical_column_with_identity

- **Contents**
- Aliases:

Returns a feature column that represents sequences of integers.

Aliases:

- `tf.compat.v1.feature_column.sequence_categorical_column_with_identity`
- `tf.compat.v2.feature_column.sequence_categorical_column_with_identity`
- `tf.feature_column.sequence_categorical_column_with_identity`

```
tf.feature_column.sequence_categorical_column_with_identity(
```

```
    key,

    num_buckets,

    default_value=None

)
```

Defined in `python/feature_column/sequence_feature_column.py`.
Pass this to `embedding_column` or `indicator_column` to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

Example:

```
watches = sequence_categorical_column_with_identity(

    'watches', num_buckets=1000)

watches_embedding = embedding_column(watches, dimension=10)

columns = [watches_embedding]



features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

sequence_feature_layer = SequenceFeatures(columns)

sequence_input, sequence_length = sequence_feature_layer(features)

sequence_length_mask = tf.sequence_mask(sequence_length)



rnn_cell = tf.keras.layers.SimpleRNNCell(hidden_size)

rnn_layer = tf.keras.layers.RNN(rnn_cell)

outputs, state = rnn_layer(sequence_input, mask=sequence_length_mask)
```

Args:

- `key`: A unique string identifying the input feature.
- `num_buckets`: Range of inputs. Namely, inputs are expected to be in the range `[0, num_buckets)`.
- `default_value`: If `None`, this column's graph operations will fail for out-of-range inputs. Otherwise, this value must be in the range `[0, num_buckets)`, and will replace out-of-range inputs.

Returns:

A `SequenceCategoricalColumn`.

Raises:

- `ValueError`: if `num_buckets` is less than one.
- `ValueError`: if `default_value` is not in range `[0, num_buckets)`.

tf.feature_column.sequence_categorical_column_with_vocabulary_file

- **Contents**
- Aliases:

A sequence of categorical terms where ids use a vocabulary file.

Aliases:

- `tf.compat.v1.feature_column.sequence_categorical_column_with_vocabulary_file`
- `tf.compat.v2.feature_column.sequence_categorical_column_with_vocabulary_file`
- `tf.feature_column.sequence_categorical_column_with_vocabulary_file`

```
tf.feature_column.sequence_categorical_column_with_vocabulary_file(
    key,
    vocabulary_file,
    vocabulary_size=None,
    num_oov_buckets=0,
    default_value=None,
    dtype=tf.dtypes.string
)
```

Defined in `python/feature_column/sequence_feature_column.py`.

Pass this to `embedding_column` or `indicator_column` to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

Example:

```
states = sequence_categorical_column_with_vocabulary_file(
    key='states', vocabulary_file='/us/states.txt', vocabulary_size=50,
    num_oov_buckets=5)
states_embedding = embedding_column(states, dimension=10)
columns = [states_embedding]
```

```
features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

sequence_feature_layer = SequenceFeatures(columns)

sequence_input, sequence_length = sequence_feature_layer(features)

sequence_length_mask = tf.sequence_mask(sequence_length)



rnn_cell = tf.keras.layers.SimpleRNNCell(hidden_size)

rnn_layer = tf.keras.layers.RNN(rnn_cell)

outputs, state = rnn_layer(sequence_input, mask=sequence_length_mask)
```

Args:

- `key`: A unique string identifying the input feature.
- `vocabulary_file`: The vocabulary file name.
- `vocabulary_size`: Number of the elements in the vocabulary. This must be no greater than length of `vocabulary_file`, if less than length, later values are ignored. If None, it is set to the length of `vocabulary_file`.
- `num_oov_buckets`: Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range `[vocabulary_size, vocabulary_size+num_oov_buckets)` based on a hash of the input value. A positive `num_oov_buckets` can not be specified with `default_value`.
- `default_value`: The integer ID value to return for out-of-vocabulary feature values, defaults to `-1`. This can not be specified with a positive `num_oov_buckets`.
- `dtype`: The type of features. Only string and integer types are supported.

Returns:

A `SequenceCategoricalColumn`.

Raises:

- `ValueError`: `vocabulary_file` is missing or cannot be opened.
- `ValueError`: `vocabulary_size` is missing or < 1.
- `ValueError`: `num_oov_buckets` is a negative integer.
- `ValueError`: `num_oov_buckets` and `default_value` are both specified.
- `ValueError`: `dtype` is neither string nor integer.

tf.feature_column.sequence_categorical_column_with_vocabulary_list

- **Contents**
- Aliases:

A sequence of categorical terms where ids use an in-memory list.

Aliases:

- `tf.compat.v1.feature_column.sequence_categorical_column_with_vocabulary_list`
- `tf.compat.v2.feature_column.sequence_categorical_column_with_vocabulary_list`

- `tf.feature_column.sequence_categorical_column_with_vocabulary_list`

```
tf.feature_column.sequence_categorical_column_with_vocabulary_list(

    key,

    vocabulary_list,

    dtype=None,

    default_value=-1,

    num_oov_buckets=0

)
```

Defined in `python/feature_column/sequence_feature_column.py`.

Pass this to `embedding_column` or `indicator_column` to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

Example:

```
colors = sequence_categorical_column_with_vocabulary_list(

    key='colors', vocabulary_list=('R', 'G', 'B', 'Y'),

    num_oov_buckets=2)

colors_embedding = embedding_column(colors, dimension=3)

columns = [colors_embedding]



features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

sequence_feature_layer = SequenceFeatures(columns)

sequence_input, sequence_length = sequence_feature_layer(features)

sequence_length_mask = tf.sequence_mask(sequence_length)



rnn_cell = tf.keras.layers.SimpleRNNCell(hidden_size)

rnn_layer = tf.keras.layers.RNN(rnn_cell)

outputs, state = rnn_layer(sequence_input, mask=sequence_length_mask)
```

Args:

- `key`: A unique string identifying the input feature.
- `vocabulary_list`: An ordered iterable defining the vocabulary. Each feature is mapped to the index of its value (if present) in `vocabulary_list`. Must be castable to `dtype`.
- `dtype`: The type of features. Only string and integer types are supported. If `None`, it will be inferred from `vocabulary_list`.
- `default_value`: The integer ID value to return for out-of-vocabulary feature values, defaults to `-1`. This can not be specified with a positive `num_oov_buckets`.
- `num_oov_buckets`: Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range `[len(vocabulary_list), len(vocabulary_list)+num_oov_buckets)` based on a hash of the input value. A positive `num_oov_buckets` can not be specified with `default_value`.

Returns:

A `SequenceCategoricalColumn`.

Raises:

- `ValueError`: if `vocabulary_list` is empty, or contains duplicate keys.
- `ValueError`: `num_oov_buckets` is a negative integer.
- `ValueError`: `num_oov_buckets` and `default_value` are both specified.
- `ValueError`: if `dtype` is not integer or string.

tf.feature_column.sequence_numeric_column

- **Contents**
- Aliases:

Returns a feature column that represents sequences of numeric data.

Aliases:

- `tf.compat.v1.feature_column.sequence_numeric_column`
- `tf.compat.v2.feature_column.sequence_numeric_column`
- `tf.feature_column.sequence_numeric_column`

```
tf.feature_column.sequence_numeric_column(

    key,

    shape=(1,),

    default_value=0.0,

    dtype=tf.dtypes.float32,

    normalizer_fn=None

)
```

Defined in `python/feature_column/sequence_feature_column.py`.

Example:

```
temperature = sequence_numeric_column('temperature')
```

```
columns = [temperature]


features = tf.io.parse_example(..., features=make_parse_example_spec(columns))
sequence_feature_layer = SequenceFeatures(columns)
sequence_input, sequence_length = sequence_feature_layer(features)
sequence_length_mask = tf.sequence_mask(sequence_length)


rnn_cell = tf.keras.layers.SimpleRNNCell(hidden_size)
rnn_layer = tf.keras.layers.RNN(rnn_cell)
outputs, state = rnn_layer(sequence_input, mask=sequence_length_mask)
```

Args:

- `key`: A unique string identifying the input features.
- `shape`: The shape of the input data per sequence id. E.g. if `shape=(2,)`, each example must contain `2 * sequence_length` values.
- `default_value`: A single value compatible with `dtype` that is used for padding the sparse data into a dense `Tensor`.
- `dtype`: The type of values.
- `normalizer_fn`: If not `None`, a function that can be used to normalize the value of the tensor after `default_value` is applied for parsing. Normalizer function takes the input `Tensor` as its argument, and returns the output `Tensor`. (e.g. lambda x: (x - 3.0) / 4.2). Please note that even though the most common use case of this function is normalization, it can be used for any kind of Tensorflow transformations.

Returns:

A `SequenceNumericColumn`.

Raises:

- `TypeError`: if any dimension in shape is not an int.
- `ValueError`: if any dimension in shape is not a positive integer.
- `ValueError`: if `dtype` is not convertible to `tf.float32`.

tf.feature_column.shared_embeddings

- **Contents**
- Aliases:

List of dense columns that convert from sparse, categorical input.

Aliases:

- `tf.compat.v2.feature_column.shared_embeddings`
- `tf.feature_column.shared_embeddings`

```
tf.feature_column.shared_embeddings(

    categorical_columns,

    dimension,

    combiner='mean',

    initializer=None,

    shared_embedding_collection_name=None,

    ckpt_to_load_from=None,

    tensor_name_in_ckpt=None,

    max_norm=None,

    trainable=True

)
```

Defined in `python/feature_column/feature_column_v2.py`.
This is similar to `embedding_column`, except that it produces a list of embedding columns that share the same embedding weights.
Use this when your inputs are sparse and of the same type (e.g. watched and impression video IDs that share the same vocabulary), and you want to convert them to a dense representation (e.g., to feed to a DNN).
Inputs must be a list of categorical columns created by any of the `categorical_column_*` function. They must all be of the same type and have the same arguments except `key`. E.g. they can be categorical_column_with_vocabulary_file with the same vocabulary_file. Some or all columns could also be weighted_categorical_column.
Here is an example embedding of two features for a DNNClassifier model:

```
watched_video_id = categorical_column_with_vocabulary_file(

    'watched_video_id', video_vocabulary_file, video_vocabulary_size)

impression_video_id = categorical_column_with_vocabulary_file(

    'impression_video_id', video_vocabulary_file, video_vocabulary_size)

columns = shared_embedding_columns(

    [watched_video_id, impression_video_id], dimension=10)
```

```
estimator = tf.estimator.DNNClassifier(feature_columns=columns, ...)


label_column = ...

def input_fn():

  features = tf.io.parse_example(

      ..., features=make_parse_example_spec(columns + [label_column]))

  labels = features.pop(label_column.name)

  return features, labels


estimator.train(input_fn=input_fn, steps=100)
```

Here is an example using `shared_embedding_columns` with model_fn:

```
def model_fn(features, ...):

  watched_video_id = categorical_column_with_vocabulary_file(

      'watched_video_id', video_vocabulary_file, video_vocabulary_size)

  impression_video_id = categorical_column_with_vocabulary_file(

      'impression_video_id', video_vocabulary_file, video_vocabulary_size)

  columns = shared_embedding_columns(

      [watched_video_id, impression_video_id], dimension=10)

  dense_tensor = input_layer(features, columns)

  # Form DNN layers, calculate loss, and return EstimatorSpec.

  ...
```

Args:

- `categorical_columns`: List of categorical columns created by a`categorical_column_with_*` function. These columns produce the sparse IDs that are inputs to the embedding lookup. All columns must be of the same type and have the same arguments except `key`. E.g. they can be categorical_column_with_vocabulary_file with the same vocabulary_file. Some or all columns could also be weighted_categorical_column.

- `dimension`: An integer specifying dimension of the embedding, must be > 0.
- `combiner`: A string specifying how to reduce if there are multiple entries in a single row. Currently 'mean', 'sqrtn' and 'sum' are supported, with 'mean' the default. 'sqrtn' often achieves good accuracy, in particular with bag-of-words columns. Each of this can be thought as example level normalizations on the column. For more information, see `tf.embedding_lookup_sparse`.
- `initializer`: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to `tf.compat.v1.truncated_normal_initializer` with mean `0.0` and standard deviation `1/sqrt(dimension)`.
- `shared_embedding_collection_name`: Optional collective name of these columns. If not given, a reasonable name will be chosen based on the names of `categorical_columns`.
- `ckpt_to_load_from`: String representing checkpoint name/pattern from which to restore column weights. Required if `tensor_name_in_ckpt` is not `None`.
- `tensor_name_in_ckpt`: Name of the `Tensor` in `ckpt_to_load_from` from which to restore the column weights. Required if `ckpt_to_load_from` is not `None`.
- `max_norm`: If not `None`, each embedding is clipped if its l2-norm is larger than this value, before combining.
- `trainable`: Whether or not the embedding is trainable. Default is True.

Returns:
A list of dense columns that converts from sparse input. The order of results follows the ordering of `categorical_columns`.

Raises:

- `ValueError`: if `dimension` not > 0.
- `ValueError`: if any of the given `categorical_columns` is of different type or has different arguments than the others.
- `ValueError`: if exactly one of `ckpt_to_load_from` and `tensor_name_in_ckpt` is specified.
- `ValueError`: if `initializer` is specified and is not callable.
- `RuntimeError`: if eager execution is enabled.

tf.feature_column.weighted_categorical_column

- **Contents**
- Aliases:

Applies weight values to a `CategoricalColumn`.

Aliases:

- `tf.compat.v1.feature_column.weighted_categorical_column`
- `tf.compat.v2.feature_column.weighted_categorical_column`
- `tf.feature_column.weighted_categorical_column`

```
tf.feature_column.weighted_categorical_column(

    categorical_column,

    weight_feature_key,

    dtype=tf.dtypes.float32

)
```

Defined in `python/feature_column/feature_column_v2.py`.
Use this when each of your sparse inputs has both an ID and a value. For example, if you're representing text documents as a collection of word frequencies, you can provide 2 parallel sparse input features ('terms' and 'frequencies' below).

Example:
Input `tf.Example` objects:

```
[

  features {

    feature {

      key: "terms"

      value {bytes_list {value: "very" value: "model"}}

    }

    feature {

      key: "frequencies"

      value {float_list {value: 0.3 value: 0.1}}

    }

  },

  features {

    feature {

      key: "terms"

      value {bytes_list {value: "when" value: "course" value: "human"}}

    }

    feature {

      key: "frequencies"

      value {float_list {value: 0.4 value: 0.1 value: 0.2}}

    }

  }
```

```
]


categorical_column = categorical_column_with_hash_bucket(

    column_name='terms', hash_bucket_size=1000)

weighted_column = weighted_categorical_column(

    categorical_column=categorical_column, weight_feature_key='frequencies')

columns = [weighted_column, ...]

features = tf.io.parse_example(..., features=make_parse_example_spec(columns))

linear_prediction, _, _ = linear_model(features, columns)
```

This assumes the input dictionary contains a `SparseTensor` for key 'terms', and a `SparseTensor` for key 'frequencies'. These 2 tensors must have the same indices and dense shape.

Args:

- `categorical_column`: A `CategoricalColumn` created by `categorical_column_with_*`functions.
- `weight_feature_key`: String key for weight values.
- `dtype`: Type of weights, such as `tf.float32`. Only float and integer weights are supported.

Returns:

A `CategoricalColumn` composed of two sparse features: one represents id, the other represents weight (value) of the id feature in that example.

Raises:

- `ValueError`: if `dtype` is not convertible to float.

Module: tf.image

- **Contents**
- Classes
- Functions

Image processing and decoding ops.

See the Images guide.

Classes

`class ResizeMethod`

Functions

`adjust_brightness(...)`: Adjust the brightness of RGB or Grayscale images.
`adjust_contrast(...)`: Adjust contrast of RGB or grayscale images.
`adjust_gamma(...)`: Performs Gamma Correction on the input image.
`adjust_hue(...)`: Adjust hue of RGB images.
`adjust_jpeg_quality(...)`: Adjust jpeg encoding quality of an RGB image.
`adjust_saturation(...)`: Adjust saturation of RGB images.
`central_crop(...)`: Crop the central region of the image(s).

combined_non_max_suppression(...): Greedily selects a subset of bounding boxes in descending order of score.
convert_image_dtype(...): Convert image to dtype, scaling its values if needed.
crop_and_resize(...): Extracts crops from the input image tensor and resizes them.
crop_to_bounding_box(...): Crops an image to a specified bounding box.
decode_and_crop_jpeg(...): Decode and Crop a JPEG-encoded image to a uint8 tensor.
decode_bmp(...): Decode the first frame of a BMP-encoded image to a uint8 tensor.
decode_gif(...): Decode the frame(s) of a GIF-encoded image to a uint8 tensor.
decode_image(...): Function for decode_bmp, decode_gif, decode_jpeg, and decode_png.
decode_jpeg(...): Decode a JPEG-encoded image to a uint8 tensor.
decode_png(...): Decode a PNG-encoded image to a uint8 or uint16 tensor.
draw_bounding_boxes(...): Draw bounding boxes on a batch of images.
encode_jpeg(...): JPEG-encode an image.
encode_png(...): PNG-encode an image.
extract_glimpse(...): Extracts a glimpse from the input tensor.
extract_jpeg_shape(...): Extract the shape information of a JPEG-encoded image.
extract_patches(...): Extract patches from images and put them in the "depth" output dimension.
flip_left_right(...): Flip an image horizontally (left to right).
flip_up_down(...): Flip an image vertically (upside down).
grayscale_to_rgb(...): Converts one or more images from Grayscale to RGB.
hsv_to_rgb(...): Convert one or more images from HSV to RGB.
image_gradients(...): Returns image gradients (dy, dx) for each color channel.
is_jpeg(...): Convenience function to check if the 'contents' encodes a JPEG image.
non_max_suppression(...): Greedily selects a subset of bounding boxes in descending order of score.
non_max_suppression_overlaps(...): Greedily selects a subset of bounding boxes in descending order of score.
non_max_suppression_padded(...): Greedily selects a subset of bounding boxes in descending order of score.
non_max_suppression_with_scores(...): Greedily selects a subset of bounding boxes in descending order of score.
pad_to_bounding_box(...): Pad image with zeros to the specified height and width.
per_image_standardization(...): Linearly scales each image in image to have mean 0 and variance 1.
psnr(...): Returns the Peak Signal-to-Noise Ratio between a and b.
random_brightness(...): Adjust the brightness of images by a random factor.
random_contrast(...): Adjust the contrast of an image or images by a random factor.
random_crop(...): Randomly crops a tensor to a given size.
random_flip_left_right(...): Randomly flip an image horizontally (left to right).
random_flip_up_down(...): Randomly flips an image vertically (upside down).
random_hue(...): Adjust the hue of RGB images by a random factor.
random_jpeg_quality(...): Randomly changes jpeg encoding quality for inducing jpeg noise.
random_saturation(...): Adjust the saturation of RGB images by a random factor.
resize(...): Resize images to size using the specified method.
resize_with_crop_or_pad(...): Crops and/or pads an image to a target width and height.
resize_with_pad(...): Resizes and pads an image to a target width and height.
rgb_to_grayscale(...): Converts one or more images from RGB to Grayscale.
rgb_to_hsv(...): Converts one or more images from RGB to HSV.
rgb_to_yiq(...): Converts one or more images from RGB to YIQ.
rgb_to_yuv(...): Converts one or more images from RGB to YUV.

`rot90(...)`: Rotate image(s) counter-clockwise by 90 degrees.
`sample_distorted_bounding_box(...)`: Generate a single randomly distorted bounding box for an image.
`sobel_edges(...)`: Returns a tensor holding Sobel edge maps.
`ssim(...)`: Computes SSIM index between img1 and img2.
`ssim_multiscale(...)`: Computes the MS-SSIM between img1 and img2.
`total_variation(...)`: Calculate and return the total variation for one or more images.
`transpose(...)`: Transpose image(s) by swapping the height and width dimension.
`yiq_to_rgb(...)`: Converts one or more images from YIQ to RGB.
`yuv_to_rgb(...)`: Converts one or more images from YUV to RGB.

tf.compat.v1.image.resize_area

Resize `images` to `size` using area interpolation.

```
tf.compat.v1.image.resize_area(

    images,

    size,

    align_corners=False,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
Input images can be of different types but output images are always float.
The range of pixel values for the output image might be slightly different from the range for the input image because of limited numerical precision. To guarantee an output range, for example `[0.0, 1.0]`, apply `tf.clip_by_value` to the output.
Each output pixel is computed by first transforming the pixel's footprint into the input tensor and then averaging the pixels that intersect the footprint. An input pixel's contribution to the average is weighted by the fraction of its area that intersects the footprint. This is the same as OpenCV's INTER_AREA.

Args:

- `images`: A `Tensor`. Must be one of the following types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `int64`, `half`, `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, the centers of the 4 corner pixels of the input and output tensors are aligned, preserving the values at the corner pixels. Defaults to false.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

tf.compat.v1.image.resize_bicubic

```
tf.compat.v1.image.resize_bicubic(
```

```
    images,

    size,

    align_corners=False,

    name=None,

    half_pixel_centers=False

)
```

Defined in python/ops/image_ops_impl.py.

tf.compat.v1.image.resize_bilinear

```
tf.compat.v1.image.resize_bilinear(

    images,

    size,

    align_corners=False,

    name=None,

    half_pixel_centers=False

)
```

Defined in python/ops/image_ops_impl.py.

tf.compat.v1.image.resize_image_with_pad

Resizes and pads an image to a target width and height.

```
tf.compat.v1.image.resize_image_with_pad(

    image,

    target_height,

    target_width,

    method=ResizeMethodV1.BILINEAR,

    align_corners=False
```

```
)
```

Defined in `python/ops/image_ops_impl.py`.
Resizes an image to a target width and height by keeping the aspect ratio the same without distortion. If the target dimensions don't match the image dimensions, the image is resized and then padded with zeroes to match requested dimensions.

Args:

- `image`: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- `target_height`: Target height.
- `target_width`: Target width.
- `method`: Method to use for resizing image. See `resize_images()`
- `align_corners`: bool. If True, the centers of the 4 corner pixels of the input and output tensors are aligned, preserving the values at the corner pixels. Defaults to `False`.

Raises:

- `ValueError`: if `target_height` or `target_width` are zero or negative.

Returns:
Resized and padded image. 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.compat.v1.image.resize_nearest_neighbor

```
tf.compat.v1.image.resize_nearest_neighbor(

    images,

    size,

    align_corners=False,

    name=None,

    half_pixel_centers=False

)
```

Defined in `python/ops/image_ops_impl.py`.

tf.image.adjust_brightness

- Contents
- Aliases:

Adjust the brightness of RGB or Grayscale images.

Aliases:

- `tf.compat.v1.image.adjust_brightness`
- `tf.compat.v2.image.adjust_brightness`
- `tf.image.adjust_brightness`

```
tf.image.adjust_brightness(

    image,

    delta

)
```

Defined in `python/ops/image_ops_impl.py`.

This is a convenience method that converts RGB images to float representation, adjusts their brightness, and then converts them 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`. `image` is converted to `float` and scaled appropriately if it is in fixed-point representation, and `delta` is converted to the same data type. 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`**: RGB image or images to adjust.
- **`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.adjust_contrast

- Contents
- Aliases:

Adjust contrast of RGB or grayscale images.

Aliases:

- `tf.compat.v1.image.adjust_contrast`
- `tf.compat.v2.image.adjust_contrast`
- `tf.image.adjust_contrast`

```
tf.image.adjust_contrast(

    images,

    contrast_factor

)
```

Defined in `python/ops/image_ops_impl.py`.

This is a convenience method that converts RGB images to float representation, adjusts their contrast, and then converts them 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 contrast-adjusted image or images.

tf.image.adjust_gamma

- Contents
- Aliases:

Performs Gamma Correction on the input image.

Aliases:

- `tf.compat.v1.image.adjust_gamma`
- `tf.compat.v2.image.adjust_gamma`
- `tf.image.adjust_gamma`

```
tf.image.adjust_gamma(
    image,
    gamma=1,
    gain=1
)
```

Defined in `python/ops/image_ops_impl.py`.

Also known as Power Law Transform. This function converts the input images at first to float representation, then transforms them pixelwise according to the equation `Out = gain * In**gamma`, and then converts the back to the original data type.

Args:

- **`image`**: RGB image or images to adjust.
- **`gamma`**: A scalar or tensor. Non negative real number.
- **`gain`**: A scalar or tensor. The constant multiplier.

Returns:

A Tensor. A Gamma-adjusted tensor of the same shape and type as `image`.

Raises:

- **`ValueError`**: If gamma is negative.

Notes:

For gamma greater than 1, the histogram will shift towards left and the output image will be darker than the input image. For gamma less than 1, the histogram will shift towards right and the output image will be brighter than the input image.

References:

[1] http://en.wikipedia.org/wiki/Gamma_correction

tf.image.adjust_hue

- Contents
- Aliases:

Adjust hue of RGB images.

Aliases:

- `tf.compat.v1.image.adjust_hue`
- `tf.compat.v2.image.adjust_hue`
- `tf.image.adjust_hue`

```
tf.image.adjust_hue(

    image,

    delta,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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(s) 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.adjust_jpeg_quality

- Contents
- Aliases:

Adjust jpeg encoding quality of an RGB image.

Aliases:

- `tf.compat.v1.image.adjust_jpeg_quality`
- `tf.compat.v2.image.adjust_jpeg_quality`
- `tf.image.adjust_jpeg_quality`

```
tf.image.adjust_jpeg_quality(

    image,

    jpeg_quality,
```

```
    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

This is a convenience method that adjusts jpeg encoding quality of an RGB image.

`image` is an RGB image. The image's encoding quality is adjusted to `jpeg_quality`. `jpeg_quality`must be in the interval `[0, 100]`.

Args:

- **`image`**: RGB image or images. Size of the last dimension must be 3.
- **`jpeg_quality`**: Python int or Tensor of type int32. jpeg encoding quality.
- **`name`**: A name for this operation (optional).

Returns:

Adjusted image(s), same shape and DType as `image`.

tf.image.adjust_saturation

- Contents
- Aliases:

Adjust saturation of RGB images.

Aliases:

- `tf.compat.v1.image.adjust_saturation`
- `tf.compat.v2.image.adjust_saturation`
- `tf.image.adjust_saturation`

```
tf.image.adjust_saturation(

    image,

    saturation_factor,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

This is a convenience method that converts RGB images to float representation, converts them 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 or images. The image saturation is adjusted by converting the images to HSV and multiplying the saturation (S) channel by `saturation_factor` and clipping. The images are 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.central_crop

- Contents
- Aliases:

Crop the central region of the image(s).

Aliases:

- `tf.compat.v1.image.central_crop`
- `tf.compat.v2.image.central_crop`
- `tf.image.central_crop`

```
tf.image.central_crop(

    image,

    central_fraction

)
```

Defined in `python/ops/image_ops_impl.py`.

Remove the outer parts of an image but retain the central region of the image along each dimension. If we specify central_fraction = 0.5, this function returns the region marked with "X" in the below diagram.

```
 --------

|        |

|  XXXX  |

|  XXXX  |

|        |   where "X" is the central 50% of the image.

 --------
```

This function works on either a single image (`image` is a 3-D Tensor), or a batch of images (`image` is a 4-D Tensor).

Args:

- **`image`**: Either a 3-D float Tensor of shape [height, width, depth], or a 4-D Tensor of shape [batch_size, height, width, depth].
- **`central_fraction`**: float (0, 1], fraction of size to crop

Raises:

- **`ValueError`**: if central_crop_fraction is not within (0, 1].

Returns:
3-D / 4-D float Tensor, as per the input.

tf.image.combined_non_max_suppression

- Contents
- Aliases:

Greedily selects a subset of bounding boxes in descending order of score.

Aliases:

- `tf.compat.v1.image.combined_non_max_suppression`
- `tf.compat.v2.image.combined_non_max_suppression`
- `tf.image.combined_non_max_suppression`

```
tf.image.combined_non_max_suppression(
    boxes,
    scores,
    max_output_size_per_class,
    max_total_size,
    iou_threshold=0.5,
    score_threshold=float('-inf'),
    pad_per_class=False,
    clip_boxes=True,
    name=None
)
```

Defined in `python/ops/image_ops_impl.py`.

This operation performs non_max_suppression on the inputs per batch, across all classes. Prunes away boxes that have high intersection-over-union (IOU) overlap with previously selected boxes. Bounding boxes are supplied as [y1, x1, y2, x2], where (y1, x1) and (y2, x2) are the coordinates of any diagonal pair of box corners and the coordinates can be provided as normalized (i.e., lying in the interval [0, 1]) or absolute. Note that this algorithm is agnostic to where the origin is in the coordinate system. Also note that this algorithm is invariant to orthogonal transformations and translations of the coordinate system; thus translating or reflections of the coordinate system result in the same boxes being selected by the algorithm. The output of this operation is the final boxes, scores and classes tensor returned after performing non_max_suppression.

Args:

- **`boxes`**: A 4-D float `Tensor` of shape `[batch_size, num_boxes, q, 4]`. If `q` is 1 then same boxes are used for all classes otherwise, if `q` is equal to number of classes, class-specific boxes are used.
- **`scores`**: A 3-D float `Tensor` of shape `[batch_size, num_boxes, num_classes]` representing a single score corresponding to each box (each row of boxes).
- **`max_output_size_per_class`**: A scalar integer `Tensor` representing the maximum number of boxes to be selected by non max suppression per class

- **`max_total_size`**: A scalar representing maximum number of boxes retained over all classes.
- **`iou_threshold`**: A float representing the threshold for deciding whether boxes overlap too much with respect to IOU.
- **`score_threshold`**: A float representing the threshold for deciding when to remove boxes based on score.
- **`pad_per_class`**: If false, the output nmsed boxes, scores and classes are padded/clipped to `max_total_size`. If true, the output nmsed boxes, scores and classes are padded to be of length `max_size_per_class`*`num_classes`, unless it exceeds `max_total_size` in which case it is clipped to `max_total_size`. Defaults to false.
- **`clip_boxes`**: If true, the coordinates of output nmsed boxes will be clipped to [0, 1]. If false, output the box coordinates as it is. Defaults to true.
- **`name`**: A name for the operation (optional).

Returns:
'nmsed_boxes': A [batch_size, max_detections, 4] float32 tensor containing the non-max suppressed boxes. 'nmsed_scores': A [batch_size, max_detections] float32 tensor containing the scores for the boxes. 'nmsed_classes': A [batch_size, max_detections] float32 tensor containing the class for boxes. 'valid_detections': A [batch_size] int32 tensor indicating the number of valid detections per batch item. Only the top valid_detections[i] entries in nms_boxes[i], nms_scores[i] and nms_class[i] are valid. The rest of the entries are zero paddings.

tf.image.convert_image_dtype

- Contents
- Aliases:
- Used in the tutorials:

Convert `image` to `dtype`, scaling its values if needed.

Aliases:

- `tf.compat.v1.image.convert_image_dtype`
- `tf.compat.v2.image.convert_image_dtype`
- `tf.image.convert_image_dtype`

```
tf.image.convert_image_dtype(
    image,
    dtype,
    saturate=False,
    name=None
)
```

Defined in `python/ops/image_ops_impl.py`.

Used in the tutorials:

- Neural style transfer

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`.

tf.image.crop_and_resize

- Contents
- Aliases:

Extracts crops from the input image tensor and resizes them.

Aliases:

- `tf.compat.v2.image.crop_and_resize`
- `tf.image.crop_and_resize`

```
tf.image.crop_and_resize(

    image,

    boxes,

    box_indices,

    crop_size,

    method='bilinear',

    extrapolation_value=0,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.
Extracts crops from the input image tensor and resizes them using bilinear sampling or nearest neighbor sampling (possibly with aspect ratio change) to a common output size specified by `crop_size`. This is more general than the `crop_to_bounding_box` op which extracts a fixed size slice from the input image and does not allow resizing or aspect ratio change.
Returns a tensor with `crops` from the input `image` at positions defined at the bounding box locations in `boxes`. The cropped boxes are all resized (with bilinear or nearest neighbor interpolation) to a

fixed`size = [crop_height, crop_width]`. The result is a 4-D tensor `[num_boxes, crop_height, crop_width, depth]`. The resizing is corner aligned. In particular, if `boxes = [[0, 0, 1, 1]]`, the method will give identical results to using `tf.compat.v1.image.resize_bilinear()` or`tf.compat.v1.image.resize_nearest_neighbor()`(depends on the `method` argument) with`align_corners=True`.

Args:

- **`image`**: A 4-D tensor of shape `[batch, image_height, image_width, depth]`. Both `image_height` and `image_width` need to be positive.
- **`boxes`**: A 2-D tensor of shape `[num_boxes, 4]`. The `i`-th row of the tensor specifies the coordinates of a box in the `box_ind[i]` image and is specified in normalized coordinates `[y1, x1, y2, x2]`. A normalized coordinate value of `y` is mapped to the image coordinate at `y * (image_height - 1)`, so as the `[0, 1]` interval of normalized image height is mapped to `[0, image_height - 1]` in image height coordinates. We do allow `y1` > `y2`, in which case the sampled crop is an up-down flipped version of the original image. The width dimension is treated similarly. Normalized coordinates outside the `[0, 1]` range are allowed, in which case we use `extrapolation_value` to extrapolate the input image values.
- **`box_indices`**: A 1-D tensor of shape `[num_boxes]` with int32 values in `[0, batch)`. The value of `box_ind[i]` specifies the image that the `i`-th box refers to.
- **`crop_size`**: A 1-D tensor of 2 elements, `size = [crop_height, crop_width]`. All cropped image patches are resized to this size. The aspect ratio of the image content is not preserved. Both `crop_height` and `crop_width` need to be positive.
- **`method`**: An optional string specifying the sampling method for resizing. It can be either `"bilinear"` or `"nearest"` and default to `"bilinear"`. Currently two sampling methods are supported: Bilinear and Nearest Neighbor.
- **`extrapolation_value`**: An optional `float`. Defaults to `0`. Value used for extrapolation, when applicable.
- **`name`**: A name for the operation (optional).

Returns:

A 4-D tensor of shape `[num_boxes, crop_height, crop_width, depth]`.

tf.image.crop_to_bounding_box

- Contents
- Aliases:

Crops an image to a specified bounding box.

Aliases:

- `tf.compat.v1.image.crop_to_bounding_box`
- `tf.compat.v2.image.crop_to_bounding_box`
- `tf.image.crop_to_bounding_box`

```
tf.image.crop_to_bounding_box(

    image,

    offset_height,

    offset_width,

    target_height,
```

```
    target_width
)
```

Defined in python/ops/image_ops_impl.py.
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`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of 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:

If `image` was 4-D, a 4-D float Tensor of shape `[batch, target_height, target_width, channels]` If `image` was 3-D, a 3-D float Tensor of shape `[target_height, target_width, channels]`

Raises:

- **`ValueError`**: If the shape of `image` is incompatible with the `offset_*` or `target_*`arguments, or either `offset_height` or `offset_width` is negative, or either `target_height`or `target_width` is not positive.

tf.image.draw_bounding_boxes

- Contents
- Aliases:

Draw bounding boxes on a batch of images.

Aliases:

- `tf.compat.v2.image.draw_bounding_boxes`
- `tf.image.draw_bounding_boxes`

```
tf.image.draw_bounding_boxes(
    images,
    boxes,
    colors,
    name=None
)
```

Defined in python/ops/image_ops_impl.py.
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 (height x width) and the bounding box is `[0.1, 0.2, 0.5, 0.9]`, the upper-left and bottom-right coordinates of the bounding box will be `(40, 10)` to `(180, 50)` (in (x,y) coordinates).
Parts of the bounding box may fall outside the image.

Args:

- **`images`**: A `Tensor`. Must be one of the following types: `float32`, `half`. 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.
- **`colors`**: A `Tensor` of type `float32`. 2-D. A list of RGBA colors to cycle through for the boxes.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has the same type as `images`.

tf.image.encode_png

- Contents
- Aliases:

PNG-encode an image.

Aliases:

- `tf.compat.v1.image.encode_png`
- `tf.compat.v2.image.encode_png`
- `tf.image.encode_png`

```
tf.image.encode_png(

    image,

    compression=-1,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
`image` is a 3-D uint8 or uint16 Tensor of shape `[height, width, channels]` where `channels` is:

- 1: for grayscale.
- 2: for grayscale + alpha.
- 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`.

tf.image.extract_glimpse

- Contents
- Aliases:

Extracts a glimpse from the input tensor.

Aliases:

- `tf.compat.v2.image.extract_glimpse`
- `tf.image.extract_glimpse`

```
tf.image.extract_glimpse(

    input,

    size,

    offsets,

    centered=True,

    normalized=True,

    noise='uniform',

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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 y, x 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.
- **noise**: An optional `string`. Defaults to `uniform`. indicates if the noise should be `uniform`(uniform distribution), `gaussian` (gaussian distribution), or `zero` (zero padding).
- **name**: A name for the operation (optional).

Returns:
A `Tensor` of type `float32`.

tf.image.extract_patches

- Contents
- Aliases:

Extract `patches` from `images` and put them in the "depth" output dimension.

Aliases:

- `tf.compat.v1.image.extract_patches`
- `tf.compat.v2.image.extract_patches`
- `tf.image.extract_patches`

```
tf.image.extract_patches(

    images,

    sizes,

    strides,

    rates,

    padding,

    name=None

)
```

Defined in `python/ops/array_ops.py`.

Args:

- **images**: A 4-D Tensor with shape `[batch, in_rows, in_cols, depth]
- **sizes**: The size of the sliding window for each dimension of `images`.
- **strides**: A 1-D Tensor of length 4. How far the centers of two consecutive patches are in the images. Must be: `[1, stride_rows, stride_cols, 1]`.
- **rates**: A 1-D Tensor of length 4. Must be: `[1, rate_rows, rate_cols, 1]`. This is the input stride, specifying how far two consecutive patch samples are in the input. Equivalent to extracting patches with `patch_sizes_eff = patch_sizes + (patch_sizes - 1) * (rates - 1)`, followed by subsampling them spatially by a factor of `rates`. This is equivalent to `rate`in dilated (a.k.a. Atrous) convolutions.

- **`padding`**: The type of padding algorithm to use. We specify the size-related attributes as: ```python ksizes = [1, ksize_rows, ksize_cols, 1] strides = [1, strides_rows, strides_cols, 1] rates = [1, rates_rows, rates_cols, 1]
- **`name`**: A name for the operation (optional).

Returns:
A 4-D Tensor. Has the same type as `images`, and with shape `[batch, out_rows, out_cols, ksize_rows * ksize_cols * depth]` containing image patches with size `ksize_rows x ksize_cols x depth` vectorized in the "depth" dimension. Note `out_rows` and `out_cols` are the dimensions of the output patches.

tf.image.flip_left_right

- Contents
- Aliases:
- Used in the tutorials:

Flip an image horizontally (left to right).

Aliases:

- `tf.compat.v1.image.flip_left_right`
- `tf.compat.v2.image.flip_left_right`
- `tf.image.flip_left_right`

```
tf.image.flip_left_right(image)
```

Defined in `python/ops/image_ops_impl.py`.

Used in the tutorials:

- Pix2Pix

Outputs the contents of `image` flipped along the width dimension.
See also `reverse()`.

Args:

- **`image`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.

Returns:
A tensor of the same type and shape as `image`.

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.flip_up_down

- Contents
- Aliases:

Flip an image vertically (upside down).

Aliases:

- `tf.compat.v1.image.flip_up_down`
- `tf.compat.v2.image.flip_up_down`
- `tf.image.flip_up_down`

```
tf.image.flip_up_down(image)
```

Defined in `python/ops/image_ops_impl.py`.

Outputs the contents of `image` flipped along the height dimension.
See also `reverse()`.

Args:

- **`image`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.

Returns:
A `Tensor` of the same type and shape as `image`.

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.grayscale_to_rgb

- Contents
- Aliases:

Converts one or more images from Grayscale to RGB.

Aliases:

- `tf.compat.v1.image.grayscale_to_rgb`
- `tf.compat.v2.image.grayscale_to_rgb`
- `tf.image.grayscale_to_rgb`

```
tf.image.grayscale_to_rgb(

    images,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.
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. The input images' last dimension must be size 1.

Args:

- **`images`**: The Grayscale tensor to convert. Last dimension must be size 1.
- **`name`**: A name for the operation (optional).

Returns:
The converted grayscale image(s).

tf.image.hsv_to_rgb

- Contents
- Aliases:

Convert one or more images from HSV to RGB.

Aliases:

- `tf.compat.v1.image.hsv_to_rgb`
- `tf.compat.v2.image.hsv_to_rgb`
- `tf.image.hsv_to_rgb`

```
tf.image.hsv_to_rgb(
```

```
    images,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
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`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. 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`. Has the same type as `images`.

tf.image.image_gradients

- Contents
- Aliases:

Returns image gradients (dy, dx) for each color channel.

Aliases:

- `tf.compat.v1.image.image_gradients`
- `tf.compat.v2.image.image_gradients`
- `tf.image.image_gradients`

```
tf.image.image_gradients(image)
```

Defined in `python/ops/image_ops_impl.py`.
Both output tensors have the same shape as the input: [batch_size, h, w, d]. The gradient values are organized so that [I(x+1, y) - I(x, y)] is in location (x, y). That means that dy will always have zeros in the last row, and dx will always have zeros in the last column.

Arguments:

- **`image`**: Tensor with shape [batch_size, h, w, d].

Returns:
Pair of tensors (dy, dx) holding the vertical and horizontal image gradients (1-step finite difference).

Raises:

- **`ValueError`**: If `image` is not a 4D tensor.

tf.image.non_max_suppression

- Contents
- Aliases:

Greedily selects a subset of bounding boxes in descending order of score.

Aliases:

- `tf.compat.v1.image.non_max_suppression`
- `tf.compat.v2.image.non_max_suppression`

- `tf.image.non_max_suppression`

```
tf.image.non_max_suppression(

    boxes,

    scores,

    max_output_size,

    iou_threshold=0.5,

    score_threshold=float('-inf'),

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Prunes away boxes that have high intersection-over-union (IOU) overlap with previously selected boxes. Bounding boxes are supplied as `[y1, x1, y2, x2]`, where `(y1, x1)` and `(y2, x2)` are the coordinates of any diagonal pair of box corners and the coordinates can be provided as normalized (i.e., lying in the interval `[0, 1]`) or absolute. Note that this algorithm is agnostic to where the origin is in the coordinate system. Note that this algorithm is invariant to orthogonal transformations and translations of the coordinate system; thus translating or reflections of the coordinate system result in the same boxes being selected by the algorithm. The output of this operation is a set of integers indexing into the input collection of bounding boxes representing the selected boxes. The bounding box coordinates corresponding to the selected indices can then be obtained using the `tf.gather`operation. For example:

```
selected_indices = tf.image.non_max_suppression(

    boxes, scores, max_output_size, iou_threshold)

selected_boxes = tf.gather(boxes, selected_indices)
```

Args:

- **`boxes`**: A 2-D float `Tensor` of shape `[num_boxes, 4]`.
- **`scores`**: A 1-D float `Tensor` of shape `[num_boxes]` representing a single score corresponding to each box (each row of boxes).
- **`max_output_size`**: A scalar integer `Tensor` representing the maximum number of boxes to be selected by non max suppression.
- **`iou_threshold`**: A float representing the threshold for deciding whether boxes overlap too much with respect to IOU.
- **`score_threshold`**: A float representing the threshold for deciding when to remove boxes based on score.
- **`name`**: A name for the operation (optional).

Returns:

- `selected_indices`: A 1-D integer `Tensor` of shape `[M]` representing the selected indices from the boxes tensor, where `M <= max_output_size`.

tf.image.non_max_suppression_overlaps

- Contents
- Aliases:

Greedily selects a subset of bounding boxes in descending order of score.

Aliases:

- `tf.compat.v1.image.non_max_suppression_overlaps`
- `tf.compat.v2.image.non_max_suppression_overlaps`
- `tf.image.non_max_suppression_overlaps`

```
tf.image.non_max_suppression_overlaps(

    overlaps,

    scores,

    max_output_size,

    overlap_threshold=0.5,

    score_threshold=float('-inf'),

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Prunes away boxes that have high overlap with previously selected boxes. N-by-n overlap values are supplied as square matrix. The output of this operation is a set of integers indexing into the input collection of bounding boxes representing the selected boxes. The bounding box coordinates corresponding to the selected indices can then be obtained using the `tf.gather` operation. For example:

```
selected_indices = tf.image.non_max_suppression_overlaps(

    overlaps, scores, max_output_size, iou_threshold)

selected_boxes = tf.gather(boxes, selected_indices)
```

Args:

- **`overlaps`**: A 2-D float `Tensor` of shape `[num_boxes, num_boxes]`.
- **`scores`**: A 1-D float `Tensor` of shape `[num_boxes]` representing a single score corresponding to each box (each row of boxes).
- **`max_output_size`**: A scalar integer `Tensor` representing the maximum number of boxes to be selected by non max suppression.

- `overlap_threshold`: A float representing the threshold for deciding whether boxes overlap too much with respect to the provided overlap values.
- `score_threshold`: A float representing the threshold for deciding when to remove boxes based on score.
- `name`: A name for the operation (optional).

Returns:

- `selected_indices`: A 1-D integer `Tensor` of shape `[M]` representing the selected indices from the overlaps tensor, where `M <= max_output_size`.

tf.image.non_max_suppression_padded

- Contents
- Aliases:

Greedily selects a subset of bounding boxes in descending order of score.

Aliases:

- `tf.compat.v1.image.non_max_suppression_padded`
- `tf.compat.v2.image.non_max_suppression_padded`
- `tf.image.non_max_suppression_padded`

```
tf.image.non_max_suppression_padded(

    boxes,

    scores,

    max_output_size,

    iou_threshold=0.5,

    score_threshold=float('-inf'),

    pad_to_max_output_size=False,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Performs algorithmically equivalent operation to tf.image.non_max_suppression, with the addition of an optional parameter which zero-pads the output to be of size `max_output_size`. The output of this operation is a tuple containing the set of integers indexing into the input collection of bounding boxes representing the selected boxes and the number of valid indices in the index set. The bounding box coordinates corresponding to the selected indices can then be obtained using the `tf.slice` and `tf.gather` operations. For example:

```
selected_indices_padded, num_valid = tf.image.non_max_suppression_padded(

    boxes, scores, max_output_size, iou_threshold,
```

```
    score_threshold, pad_to_max_output_size=True)

selected_indices = tf.slice(

    selected_indices_padded, tf.constant([0]), num_valid)

selected_boxes = tf.gather(boxes, selected_indices)
```

Args:

- **`boxes`**: A 2-D float `Tensor` of shape `[num_boxes, 4]`.
- **`scores`**: A 1-D float `Tensor` of shape `[num_boxes]` representing a single score corresponding to each box (each row of boxes).
- **`max_output_size`**: A scalar integer `Tensor` representing the maximum number of boxes to be selected by non max suppression.
- **`iou_threshold`**: A float representing the threshold for deciding whether boxes overlap too much with respect to IOU.
- **`score_threshold`**: A float representing the threshold for deciding when to remove boxes based on score.
- **`pad_to_max_output_size`**: bool. If True, size of `selected_indices` output is padded to `max_output_size`.
- **`name`**: A name for the operation (optional).

Returns:

- **`selected_indices`**: A 1-D integer `Tensor` of shape `[M]` representing the selected indices from the boxes tensor, where `M <= max_output_size`.
- **`valid_outputs`**: A scalar integer `Tensor` denoting how many elements in `selected_indices`are valid. Valid elements occur first, then padding.

tf.image.non_max_suppression_with_scores

- Contents
- Aliases:

Greedily selects a subset of bounding boxes in descending order of score.

Aliases:

- `tf.compat.v1.image.non_max_suppression_with_scores`
- `tf.compat.v2.image.non_max_suppression_with_scores`
- `tf.image.non_max_suppression_with_scores`

```
tf.image.non_max_suppression_with_scores(

    boxes,

    scores,

    max_output_size,

    iou_threshold=0.5,

    score_threshold=float('-inf'),
```

```
    soft_nms_sigma=0.0,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Prunes away boxes that have high intersection-over-union (IOU) overlap with previously selected boxes. Bounding boxes are supplied as `[y1, x1, y2, x2]`, where `(y1, x1)` and `(y2, x2)` are the coordinates of any diagonal pair of box corners and the coordinates can be provided as normalized (i.e., lying in the interval `[0, 1]`) or absolute. Note that this algorithm is agnostic to where the origin is in the coordinate system. Note that this algorithm is invariant to orthogonal transformations and translations of the coordinate system; thus translating or reflections of the coordinate system result in the same boxes being selected by the algorithm. The output of this operation is a set of integers indexing into the input collection of bounding boxes representing the selected boxes. The bounding box coordinates corresponding to the selected indices can then be obtained using the `tf.gather`operation. For example:

```
selected_indices, selected_scores = tf.image.non_max_suppression_v2(

    boxes, scores, max_output_size, iou_threshold=1.0, score_threshold=0.1,

    soft_nms_sigma=0.5)

selected_boxes = tf.gather(boxes, selected_indices)
```

This function generalizes the `tf.image.non_max_suppression` op by also supporting a Soft-NMS (with Gaussian weighting) mode (c.f. Bodla et al, https://arxiv.org/abs/1704.04503) where boxes reduce the score of other overlapping boxes instead of directly causing them to be pruned. Consequently, in contrast to `tf.image.non_max_suppression`,`tf.image.non_max_suppression_v2` returns the new scores of each input box in the second output, `selected_scores`.

To enable this Soft-NMS mode, set the `soft_nms_sigma` parameter to be larger than 0. When `soft_nms_sigma` equals 0, the behavior of `tf.image.non_max_suppression_v2` is identical to that of `tf.image.non_max_suppression` (except for the extra output) both in function and in running time.

Args:

- **`boxes`**: A 2-D float `Tensor` of shape `[num_boxes, 4]`.
- **`scores`**: A 1-D float `Tensor` of shape `[num_boxes]` representing a single score corresponding to each box (each row of boxes).
- **`max_output_size`**: A scalar integer `Tensor` representing the maximum number of boxes to be selected by non max suppression.
- **`iou_threshold`**: A float representing the threshold for deciding whether boxes overlap too much with respect to IOU.
- **`score_threshold`**: A float representing the threshold for deciding when to remove boxes based on score.
- **`soft_nms_sigma`**: A scalar float representing the Soft NMS sigma parameter; See Bodla et al, https://arxiv.org/abs/1704.04503). When `soft_nms_sigma=0.0` (which is default), we fall back to standard (hard) NMS.

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

Returns:

- **`selected_indices`**: A 1-D integer `Tensor` of shape `[M]` representing the selected indices from the boxes tensor, where `M <= max_output_size`.
- **`selected_scores`**: A 1-D float tensor of shape `[M]` representing the corresponding scores for each selected box, where `M <= max_output_size`. Scores only differ from corresponding input scores when using Soft NMS (i.e. when `soft_nms_sigma>0`)

tf.image.pad_to_bounding_box

- Contents
- Aliases:

Pad `image` with zeros to the specified `height` and `width`.

Aliases:

- `tf.compat.v1.image.pad_to_bounding_box`
- `tf.compat.v2.image.pad_to_bounding_box`
- `tf.image.pad_to_bounding_box`

```
tf.image.pad_to_bounding_box(

    image,

    offset_height,

    offset_width,

    target_height,

    target_width

)
```

Defined in `python/ops/image_ops_impl.py`.

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`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of 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:

If `image` was 4-D, a 4-D float Tensor of shape `[batch, target_height, target_width, channels]` If `image` was 3-D, a 3-D float Tensor of shape `[target_height, target_width, channels]`

Raises:

- **`ValueError`**: If the shape of `image` is incompatible with the `offset_*` or `target_*`arguments, or either `offset_height` or `offset_width` is negative.

tf.image.per_image_standardization

- Contents
- Aliases:

Linearly scales each image in `image` to have mean 0 and variance 1.

Aliases:

- `tf.compat.v1.image.per_image_standardization`
- `tf.compat.v2.image.per_image_standardization`
- `tf.image.per_image_standardization`

```
tf.image.per_image_standardization(image)
```

Defined in `python/ops/image_ops_impl.py`.

For each 3-D image `x` in `image`, computes `(x - mean) / adjusted_stddev`, where

- `mean` is the average of all values in `x`
- `adjusted_stddev = max(stddev, 1.0/sqrt(N))` is capped away from 0 to protect against division by 0 when handling uniform images
- `N` is the number of elements in `x`
- `stddev` is the standard deviation of all values in `x`

Args:

- **`image`**: An n-D Tensor with at least 3 dimensions, the last 3 of which are the dimensions of each image.

Returns:

A `Tensor` with same shape and dtype as `image`.

Raises:

- **`ValueError`**: if the shape of 'image' is incompatible with this function.

tf.image.psnr

- Contents
- Aliases:

Returns the Peak Signal-to-Noise Ratio between a and b.

Aliases:

- `tf.compat.v1.image.psnr`
- `tf.compat.v2.image.psnr`
- `tf.image.psnr`

```
tf.image.psnr(

    a,

    b,

    max_val,
```

```
    name=None
)
```

Defined in `python/ops/image_ops_impl.py`.
This is intended to be used on signals (or images). Produces a PSNR value for each image in batch.
The last three dimensions of input are expected to be [height, width, depth].

Example:

```
    # Read images from file.
    im1 = tf.decode_png('path/to/im1.png')
    im2 = tf.decode_png('path/to/im2.png')
    # Compute PSNR over tf.uint8 Tensors.
    psnr1 = tf.image.psnr(im1, im2, max_val=255)


    # Compute PSNR over tf.float32 Tensors.
    im1 = tf.image.convert_image_dtype(im1, tf.float32)
    im2 = tf.image.convert_image_dtype(im2, tf.float32)
    psnr2 = tf.image.psnr(im1, im2, max_val=1.0)
    # psnr1 and psnr2 both have type tf.float32 and are almost equal.
```

Arguments:

- **`a`**: First set of images.
- **`b`**: Second set of images.
- **`max_val`**: The dynamic range of the images (i.e., the difference between the maximum the and minimum allowed values).
- **`name`**: Namespace to embed the computation in.

Returns:
The scalar PSNR between a and b. The returned tensor has type `tf.float32` and shape [batch_size, 1].

tf.image.random_brightness

- Contents
- Aliases:

Adjust the brightness of images by a random factor.

Aliases:

- `tf.compat.v1.image.random_brightness`

- `tf.compat.v2.image.random_brightness`
- `tf.image.random_brightness`

```
tf.image.random_brightness(
    image,
    max_delta,
    seed=None
)
```

Defined in `python/ops/image_ops_impl.py`.

Equivalent to `adjust_brightness()` using a `delta` randomly picked in the interval `[-max_delta, max_delta)`.

Args:

- **`image`**: An image or images to adjust.
- **`max_delta`**: float, must be non-negative.
- **`seed`**: A Python integer. Used to create a random seed. See `tf.compat.v1.set_random_seed`for behavior.

Returns:

The brightness-adjusted image(s).

Raises:

- **`ValueError`**: if `max_delta` is negative.

tf.image.random_contrast

- Contents
- Aliases:

Adjust the contrast of an image or images by a random factor.

Aliases:

- `tf.compat.v1.image.random_contrast`
- `tf.compat.v2.image.random_contrast`
- `tf.image.random_contrast`

```
tf.image.random_contrast(
    image,
    lower,
    upper,
    seed=None
)
```

Defined in `python/ops/image_ops_impl.py`.

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 `tf.compat.v1.set_random_seed`for behavior.

Returns:

The contrast-adjusted image(s).

Raises:

- **`ValueError`**: if `upper <= lower` or if `lower < 0`.

tf.image.random_crop

- Contents
- Aliases:
- Used in the tutorials:

Randomly crops a tensor to a given size.

Aliases:

- `tf.compat.v1.image.random_crop`
- `tf.compat.v1.random_crop`
- `tf.compat.v2.image.random_crop`
- `tf.image.random_crop`

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

Defined in `python/ops/random_ops.py`.

Used in the tutorials:

- Pix2Pix

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 `tf.compat.v1.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.image.random_flip_left_right

- Contents
- Aliases:

Randomly flip an image horizontally (left to right).

Aliases:

- `tf.compat.v1.image.random_flip_left_right`
- `tf.compat.v2.image.random_flip_left_right`
- `tf.image.random_flip_left_right`

```
tf.image.random_flip_left_right(
    image,
    seed=None
)
```

Defined in `python/ops/image_ops_impl.py`.

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`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`seed`**: A Python integer. Used to create a random seed. See `tf.compat.v1.set_random_seed`for behavior.

Returns:

A tensor of the same type and shape as `image`.

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.random_flip_up_down

- Contents
- Aliases:

Randomly flips an image vertically (upside down).

Aliases:

- `tf.compat.v1.image.random_flip_up_down`
- `tf.compat.v2.image.random_flip_up_down`
- `tf.image.random_flip_up_down`

```
tf.image.random_flip_up_down(
```

```
    image,

    seed=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`seed`**: A Python integer. Used to create a random seed. See `tf.compat.v1.set_random_seed`for behavior.

Returns:

A tensor of the same type and shape as `image`.

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.random_hue

- Contents
- Aliases:

Adjust the hue of RGB images by a random factor.

Aliases:

- `tf.compat.v1.image.random_hue`
- `tf.compat.v2.image.random_hue`
- `tf.image.random_hue`

```
tf.image.random_hue(

    image,

    max_delta,

    seed=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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:
Adjusted image(s), same shape and DType as `image`.

Raises:
- **ValueError**: if `max_delta` is invalid.

tf.image.random_jpeg_quality

- Contents
- Aliases:

Randomly changes jpeg encoding quality for inducing jpeg noise.

Aliases:
- `tf.compat.v1.image.random_jpeg_quality`
- `tf.compat.v2.image.random_jpeg_quality`
- `tf.image.random_jpeg_quality`

```
tf.image.random_jpeg_quality(
    image,
    min_jpeg_quality,
    max_jpeg_quality,
    seed=None
)
```

Defined in `python/ops/image_ops_impl.py`.
`min_jpeg_quality` must be in the interval `[0, 100]` and less than `max_jpeg_quality`.`max_jpeg_quality` must be in the interval `[0, 100]`.

Args:
- **image**: RGB image or images. Size of the last dimension must be 3.
- **min_jpeg_quality**: Minimum jpeg encoding quality to use.
- **max_jpeg_quality**: Maximum jpeg encoding quality to use.
- **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 `min_jpeg_quality` or `max_jpeg_quality` is invalid.

tf.image.random_saturation

- Contents
- Aliases:

Adjust the saturation of RGB images by a random factor.

Aliases:

- `tf.compat.v1.image.random_saturation`
- `tf.compat.v2.image.random_saturation`
- `tf.image.random_saturation`

```
tf.image.random_saturation(

    image,

    lower,

    upper,

    seed=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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.resize

- Contents
- Aliases:
- Used in the tutorials:

Resize `images` to `size` using the specified `method`.

Aliases:

- `tf.compat.v2.image.resize`
- `tf.image.resize`

```
tf.image.resize(

    images,

    size,
```

```
    method=ResizeMethod.BILINEAR,

    preserve_aspect_ratio=False,

    antialias=False,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Used in the tutorials:

- Image Captioning with Attention
- Load images with tf.data
- Neural style transfer
- Pix2Pix
- Transfer Learning Using Pretrained ConvNets

Resized images will be distorted if their original aspect ratio is not the same as `size`. To avoid distortions see `tf.image.resize_with_pad`.

When 'antialias' is true, the sampling filter will anti-alias the input image as well as interpolate. When downsampling an image with anti-aliasing the sampling filter kernel is scaled in order to properly anti-alias the input image signal. 'antialias' has no effect when upsampling an image.

- **`bilinear`**: Bilinear interpolation. If 'antialias' is true, becomes a hat/tent filter function with radius 1 when downsampling.
- **`lanczos3`**: Lanczos kernel with radius 3. High-quality practical filter but may have some ringing especially on synthetic images.
- **`lanczos5`**: Lanczos kernel with radius 5. Very-high-quality filter but may have stronger ringing.
- **`bicubic`**: Cubic interpolant of Keys. Equivalent to Catmull-Rom kernel. Reasonably good quality and faster than Lanczos3Kernel, particularly when upsampling.
- **`gaussian`**: Gaussian kernel with radius 3, sigma = 1.5 / 3.0.
- **`nearest`**: Nearest neighbor interpolation. 'antialias' has no effect when used with nearest neighbor interpolation.
- **`area`**: Anti-aliased resampling with area interpolation. 'antialias' has no effect when used with area interpolation; it always anti-aliases.
- **`mitchellcubic`**: Mitchell-Netravali Cubic non-interpolating filter. For synthetic images (especially those lacking proper prefiltering), less ringing than Keys cubic kernel but less sharp.

Note that near image edges the filtering kernel may be partially outside the image boundaries. For these pixels, only input pixels inside the image will be included in the filter sum, and the output value will be appropriately normalized.

The return value has the same type as `images` if `method` is `ResizeMethod.NEAREST_NEIGHBOR`. Otherwise, the return value has type `float32`.

Args:

- **`images`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`size`**: A 1-D int32 Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- **`method`**: ResizeMethod. Defaults to `bilinear`.
- **`preserve_aspect_ratio`**: Whether to preserve the aspect ratio. If this is set, then `images` will be resized to a size that fits in `size` while preserving the aspect ratio of the original image. Scales up the image if `size` is bigger than the current size of the `image`. Defaults to False.

- **`antialias`**: Whether to use an anti-aliasing filter when downsampling an image.
- **`name`**: A name for this operation (optional).

Raises:

- **`ValueError`**: if the shape of `images` is incompatible with the shape arguments to this function
- **`ValueError`**: if `size` has invalid shape or type.
- **`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.ResizeMethod

Class `ResizeMethod`

Aliases:

- Class `tf.compat.v2.image.ResizeMethod`
- Class `tf.image.ResizeMethod`

Defined in `python/ops/image_ops_impl.py`.

Class Members

- `AREA = 'area'`
- `BICUBIC = 'bicubic'`
- `BILINEAR = 'bilinear'`
- `GAUSSIAN = 'gaussian'`
- `LANCZOS3 = 'lanczos3'`
- `LANCZOS5 = 'lanczos5'`
- `MITCHELLCUBIC = 'mitchellcubic'`
- `NEAREST_NEIGHBOR = 'nearest'`

tf.image.resize_with_crop_or_pad

Crops and/or pads an image to a target width and height.

Aliases:

- `tf.compat.v1.image.resize_image_with_crop_or_pad`
- `tf.compat.v1.image.resize_with_crop_or_pad`
- `tf.compat.v2.image.resize_with_crop_or_pad`
- `tf.image.resize_with_crop_or_pad`

```
tf.image.resize_with_crop_or_pad(

    image,

    target_height,
```

```
    target_width
)
```

Defined in `python/ops/image_ops_impl.py`.

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`**: 4-D Tensor of shape `[batch, height, width, channels]` or 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. 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_with_pad

- Contents
- Aliases:

Resizes and pads an image to a target width and height.

Aliases:

- `tf.compat.v2.image.resize_with_pad`
- `tf.image.resize_with_pad`

```
tf.image.resize_with_pad(
    image,
    target_height,
    target_width,
    method=ResizeMethod.BILINEAR,
    antialias=False
)
```

Defined in `python/ops/image_ops_impl.py`.

Resizes an image to a target width and height by keeping the aspect ratio the same without distortion. If the target dimensions don't match the image dimensions, the image is resized and then padded with zeroes to match requested dimensions.

Args:

- **`image`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`target_height`**: Target height.
- **`target_width`**: Target width.
- **`method`**: Method to use for resizing image. See `image.resize()`
- **`antialias`**: Whether to use anti-aliasing when resizing. See 'image.resize()'.

Raises:

- **`ValueError`**: if `target_height` or `target_width` are zero or negative.

Returns:

Resized and padded image. 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.rgb_to_grayscale

- Contents
- Aliases:

Converts one or more images from RGB to Grayscale.

Aliases:

- `tf.compat.v1.image.rgb_to_grayscale`
- `tf.compat.v2.image.rgb_to_grayscale`
- `tf.image.rgb_to_grayscale`

```
tf.image.rgb_to_grayscale(

    images,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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

Returns:

The converted grayscale image(s).

tf.image.rgb_to_hsv

- Contents
- Aliases:

Converts one or more images from RGB to HSV.

Aliases:

- `tf.compat.v1.image.rgb_to_hsv`
- `tf.compat.v2.image.rgb_to_hsv`
- `tf.image.rgb_to_hsv`

```
tf.image.rgb_to_hsv(

    images,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.

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`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. 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`. Has the same type as `images`.

tf.image.rgb_to_yiq

- Contents
- Aliases:

Converts one or more images from RGB to YIQ.

Aliases:

- `tf.compat.v1.image.rgb_to_yiq`
- `tf.compat.v2.image.rgb_to_yiq`
- `tf.image.rgb_to_yiq`

```
tf.image.rgb_to_yiq(images)
```

Defined in `python/ops/image_ops_impl.py`.

Outputs a tensor of the same shape as the `images` tensor, containing the YIQ value of the pixels. The output is only well defined if the value in images are in [0,1].

Args:

- **`images`**: 2-D or higher rank. Image data to convert. Last dimension must be size 3.

Returns:

- **`images`**: tensor with the same shape as `images`.

tf.image.rgb_to_yuv

- Contents

- Aliases:

Converts one or more images from RGB to YUV.

Aliases:

- `tf.compat.v1.image.rgb_to_yuv`
- `tf.compat.v2.image.rgb_to_yuv`
- `tf.image.rgb_to_yuv`

```
tf.image.rgb_to_yuv(images)
```

Defined in `python/ops/image_ops_impl.py`.

Outputs a tensor of the same shape as the `images` tensor, containing the YUV value of the pixels. The output is only well defined if the value in images are in [0,1].

Args:

- **`images`**: 2-D or higher rank. Image data to convert. Last dimension must be size 3.

Returns:

- **`images`**: tensor with the same shape as `images`.

tf.image.rot90

- Contents
- Aliases:

Rotate image(s) counter-clockwise by 90 degrees.

Aliases:

- `tf.compat.v1.image.rot90`
- `tf.compat.v2.image.rot90`
- `tf.image.rot90`

```
tf.image.rot90(

    image,

    k=1,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Args:

- **`image`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`k`**: A scalar integer. The number of times the image is rotated by 90 degrees.
- **`name`**: A name for this operation (optional).

Returns:

A rotated tensor of the same type and shape as `image`.

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.sample_distorted_bounding_box

- Contents
- Aliases:

Generate a single randomly distorted bounding box for an image.

Aliases:

- `tf.compat.v2.image.sample_distorted_bounding_box`
- `tf.image.sample_distorted_bounding_box`

```
tf.image.sample_distorted_bounding_box(

    image_size,

    bounding_boxes,

    seed=0,

    min_object_covered=0.1,

    aspect_ratio_range=None,

    area_range=None,

    max_attempts=None,

    use_image_if_no_bounding_boxes=None,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

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_boxes` 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,
        min_object_covered=0.1)

    # 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.compat.v1.summary.image('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 `seed` is set to non-zero, the random number generator is seeded by the given `seed`. Otherwise, it is seeded by a random seed.
- **`min_object_covered`**: A Tensor of type `float32`. Defaults to `0.1`. The cropped area of the image must contain at least this fraction of any bounding box supplied. The value of this parameter should be non-negative. In the case of 0, the cropped area does not need to overlap any of the bounding boxes 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 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`.

tf.image.sobel_edges

- Contents
- Aliases:
- Used in the tutorials:

Returns a tensor holding Sobel edge maps.

Aliases:

- `tf.compat.v1.image.sobel_edges`
- `tf.compat.v2.image.sobel_edges`
- `tf.image.sobel_edges`

```
tf.image.sobel_edges(image)
```

Defined in `python/ops/image_ops_impl.py`.

Used in the tutorials:

- Neural style transfer

Arguments:

- **`image`**: Image tensor with shape [batch_size, h, w, d] and type float32 or float64. The image(s) must be 2x2 or larger.

Returns:
Tensor holding edge maps for each channel. Returns a tensor with shape [batch_size, h, w, d, 2] where the last two dimensions hold [[dy[0], dx[0]], [dy[1], dx[1]], ..., [dy[d-1], dx[d-1]]] calculated using the Sobel filter.

tf.image.ssim

- Contents
- Aliases:

Computes SSIM index between img1 and img2.

Aliases:

- `tf.compat.v1.image.ssim`
- `tf.compat.v2.image.ssim`
- `tf.image.ssim`

```
tf.image.ssim(

    img1,

    img2,
```

```
    max_val,

    filter_size=11,

    filter_sigma=1.5,

    k1=0.01,

    k2=0.03

)
```

Defined in `python/ops/image_ops_impl.py`.

This function is based on the standard SSIM implementation from: Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing.

Note: The true SSIM is only defined on grayscale. This function does not perform any colorspace transform. (If input is already YUV, then it will compute YUV SSIM average.)

Details:

- 11x11 Gaussian filter of width 1.5 is used.
- k1 = 0.01, k2 = 0.03 as in the original paper.

The image sizes must be at least 11x11 because of the filter size.

Example:

```
    # Read images from file.

    im1 = tf.decode_png('path/to/im1.png')

    im2 = tf.decode_png('path/to/im2.png')

    # Compute SSIM over tf.uint8 Tensors.

    ssim1 = tf.image.ssim(im1, im2, max_val=255, filter_size=11,

                          filter_sigma=1.5, k1=0.01, k2=0.03)



    # Compute SSIM over tf.float32 Tensors.

    im1 = tf.image.convert_image_dtype(im1, tf.float32)

    im2 = tf.image.convert_image_dtype(im2, tf.float32)

    ssim2 = tf.image.ssim(im1, im2, max_val=1.0, filter_size=11,

                          filter_sigma=1.5, k1=0.01, k2=0.03)
```

```
    # ssim1 and ssim2 both have type tf.float32 and are almost equal.
```

Args:

- `img1`: First image batch.
- `img2`: Second image batch.
- `max_val`: The dynamic range of the images (i.e., the difference between the maximum the and minimum allowed values).
- `filter_size`: Default value 11 (size of gaussian filter).
- `filter_sigma`: Default value 1.5 (width of gaussian filter).
- `k1`: Default value 0.01
- `k2`: Default value 0.03 (SSIM is less sensitivity to K2 for lower values, so it would be better if we taken the values in range of 0< K2 <0.4).

Returns:

A tensor containing an SSIM value for each image in batch. Returned SSIM values are in range (-1, 1], when pixel values are non-negative. Returns a tensor with shape: broadcast(img1.shape[:-3], img2.shape[:-3]).

tf.image.ssim_multiscale

- Contents
- Aliases:

Computes the MS-SSIM between img1 and img2.

Aliases:

- `tf.compat.v1.image.ssim_multiscale`
- `tf.compat.v2.image.ssim_multiscale`
- `tf.image.ssim_multiscale`

```
tf.image.ssim_multiscale(
    img1,
    img2,
    max_val,
    power_factors=_MSSSIM_WEIGHTS,
    filter_size=11,
    filter_sigma=1.5,
    k1=0.01,
    k2=0.03
)
```

Defined in `python/ops/image_ops_impl.py`.

This function assumes that `img1` and `img2` are image batches, i.e. the last three dimensions are [height, width, channels].
Note: The true SSIM is only defined on grayscale. This function does not perform any colorspace transform. (If input is already YUV, then it will compute YUV SSIM average.)
Original paper: Wang, Zhou, Eero P. Simoncelli, and Alan C. Bovik. "Multiscale structural similarity for image quality assessment." Signals, Systems and Computers, 2004.

Arguments:

- **`img1`**: First image batch.
- **`img2`**: Second image batch. Must have the same rank as img1.
- **`max_val`**: The dynamic range of the images (i.e., the difference between the maximum the and minimum allowed values).
- **`power_factors`**: Iterable of weights for each of the scales. The number of scales used is the length of the list. Index 0 is the unscaled resolution's weight and each increasing scale corresponds to the image being downsampled by 2. Defaults to (0.0448, 0.2856, 0.3001, 0.2363, 0.1333), which are the values obtained in the original paper.
- **`filter_size`**: Default value 11 (size of gaussian filter).
- **`filter_sigma`**: Default value 1.5 (width of gaussian filter).
- **`k1`**: Default value 0.01
- **`k2`**: Default value 0.03 (SSIM is less sensitivity to K2 for lower values, so it would be better if we taken the values in range of 0< K2 <0.4).

Returns:
A tensor containing an MS-SSIM value for each image in batch. The values are in range [0, 1]. Returns a tensor with shape: broadcast(img1.shape[:-3], img2.shape[:-3]).

tf.image.total_variation

- Contents
- Aliases:

Calculate and return the total variation for one or more images.

Aliases:

- `tf.compat.v1.image.total_variation`
- `tf.compat.v2.image.total_variation`
- `tf.image.total_variation`

```
tf.image.total_variation(

    images,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.
The total variation is the sum of the absolute differences for neighboring pixel-values in the input images. This measures how much noise is in the images.
This can be used as a loss-function during optimization so as to suppress noise in images. If you have a batch of images, then you should calculate the scalar loss-value as the sum: `loss = tf.reduce_sum(tf.image.total_variation(images))`
This implements the anisotropic 2-D version of the formula described here:
https://en.wikipedia.org/wiki/Total_variation_denoising

Args:

- **`images`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`name`**: A name for the operation (optional).

Raises:

- **`ValueError`**: if images.shape is not a 3-D or 4-D vector.

Returns:

The total variation of `images`.

If `images` was 4-D, return a 1-D float Tensor of shape `[batch]` with the total variation for each image in the batch. If `images` was 3-D, return a scalar float with the total variation for that image.

tf.image.transpose

- Contents
- Aliases:

Transpose image(s) by swapping the height and width dimension.

Aliases:

- `tf.compat.v1.image.transpose`
- `tf.compat.v1.image.transpose_image`
- `tf.compat.v2.image.transpose`
- `tf.image.transpose`

```
tf.image.transpose(

    image,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Args:

- **`image`**: 4-D Tensor of shape `[batch, height, width, channels]` or 3-D Tensor of shape `[height, width, channels]`.
- **`name`**: A name for this operation (optional).

Returns:

If `image` was 4-D, a 4-D float Tensor of shape `[batch, width, height, channels]` If `image` was 3-D, a 3-D float Tensor of shape `[width, height, channels]`

Raises:

- **`ValueError`**: if the shape of `image` not supported.

tf.image.yiq_to_rgb

- Contents
- Aliases:

Converts one or more images from YIQ to RGB.

Aliases:

- `tf.compat.v1.image.yiq_to_rgb`
- `tf.compat.v2.image.yiq_to_rgb`

- `tf.image.yiq_to_rgb`

```
tf.image.yiq_to_rgb(images)
```

Defined in `python/ops/image_ops_impl.py`.
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 Y value in images are in [0,1], I value are in [-0.5957,0.5957] and Q value are in [-0.5226,0.5226].

Args:
- **`images`**: 2-D or higher rank. Image data to convert. Last dimension must be size 3.

Returns:
- **`images`**: tensor with the same shape as `images`.

tf.image.yuv_to_rgb

- Contents
- Aliases:

Converts one or more images from YUV to RGB.

Aliases:
- `tf.compat.v1.image.yuv_to_rgb`
- `tf.compat.v2.image.yuv_to_rgb`
- `tf.image.yuv_to_rgb`

```
tf.image.yuv_to_rgb(images)
```

Defined in `python/ops/image_ops_impl.py`.
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 Y value in images are in [0,1], U and V value are in [-0.5,0.5].

Args:
- **`images`**: 2-D or higher rank. Image data to convert. Last dimension must be size 3.

Returns:
- **`images`**: tensor with the same shape as `images`.

Module: tf.io

- **Contents**
- Modules
- Classes
- Functions

Public API for tf.io namespace.

Modules

`gfile` module: Public API for tf.io.gfile namespace.

Classes

`class FixedLenFeature`: Configuration for parsing a fixed-length input feature.
`class FixedLenSequenceFeature`: Configuration for parsing a variable-length input feature into a `Tensor`.
`class SparseFeature`: Configuration for parsing a sparse input feature from an `Example`.
`class TFRecordOptions`: Options used for manipulating TFRecord files.
`class TFRecordWriter`: A class to write records to a TFRecords file.

`class VarLenFeature`: Configuration for parsing a variable-length input feature.

Functions

`decode_and_crop_jpeg(...)`: Decode and Crop a JPEG-encoded image to a uint8 tensor.
`decode_base64(...)`: Decode web-safe base64-encoded strings.
`decode_bmp(...)`: Decode the first frame of a BMP-encoded image to a uint8 tensor.
`decode_compressed(...)`: Decompress strings.
`decode_csv(...)`: Convert CSV records to tensors. Each column maps to one tensor.
`decode_gif(...)`: Decode the frame(s) of a GIF-encoded image to a uint8 tensor.
`decode_image(...)`: Function for `decode_bmp`, `decode_gif`, `decode_jpeg`, and `decode_png`.
`decode_jpeg(...)`: Decode a JPEG-encoded image to a uint8 tensor.
`decode_json_example(...)`: Convert JSON-encoded Example records to binary protocol buffer strings.
`decode_png(...)`: Decode a PNG-encoded image to a uint8 or uint16 tensor.
`decode_proto(...)`: The op extracts fields from a serialized protocol buffers message into tensors.
`decode_raw(...)`: Convert raw byte strings into tensors.
`deserialize_many_sparse(...)`: Deserialize and concatenate `SparseTensors` from a serialized minibatch.
`encode_base64(...)`: Encode strings into web-safe base64 format.
`encode_jpeg(...)`: JPEG-encode an image.
`encode_proto(...)`: The op serializes protobuf messages provided in the input tensors.
`extract_jpeg_shape(...)`: Extract the shape information of a JPEG-encoded image.
`is_jpeg(...)`: Convenience function to check if the 'contents' encodes a JPEG image.
`match_filenames_once(...)`: Save the list of files matching pattern, so it is only computed once.
`matching_files(...)`: Returns the set of files matching one or more glob patterns.
`parse_example(...)`: Parses `Example` protos into a `dict` of tensors.
`parse_sequence_example(...)`: Parses a batch of `SequenceExample` protos.
`parse_single_example(...)`: Parses a single `Example` proto.
`parse_single_sequence_example(...)`: Parses a single `SequenceExample` proto.
`parse_tensor(...)`: Transforms a serialized tensorflow.TensorProto proto into a Tensor.
`read_file(...)`: Reads and outputs the entire contents of the input filename.
`serialize_many_sparse(...)`: Serialize `N`-minibatch `SparseTensor` into an `[N, 3] Tensor`.
`serialize_sparse(...)`: Serialize a `SparseTensor` into a 3-vector (1-D `Tensor`) object.
`serialize_tensor(...)`: Transforms a Tensor into a serialized TensorProto proto.
`write_file(...)`: Writes contents to the file at input filename. Creates file and recursively
`write_graph(...)`: Writes a graph proto to a file.

tf.compat.v1.io.TFRecordCompressionType

- Contents
- Class TFRecordCompressionType
  - Aliases:
- Class Members

Class `TFRecordCompressionType`

The type of compression for the record.

Aliases:

- Class `tf.compat.v1.io.TFRecordCompressionType`
- Class `tf.compat.v1.python_io.TFRecordCompressionType`

Defined in `python/lib/io/tf_record.py`.

Class Members

- `GZIP = 2`
- `NONE = 0`
- `ZLIB = 1`

tf.compat.v1.io.tf_record_iterator

- Contents
- Aliases:

An iteraror that read the records from a TFRecords file. (deprecated)

Aliases:

`tf.compat.v1.io.tf_record_iterator`
`tf.compat.v1.python_io.tf_record_iterator`

```
tf.compat.v1.io.tf_record_iterator(

    path,

    options=None

)
```

Defined in `python/lib/io/tf_record.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use eager execution and: `tf.data.TFRecordDataset(path)`

Args:

`path`: The path to the TFRecords file.
`options`: (optional) A TFRecordOptions object.

Yields: Strings.

Raises: **`IOError`**: If `path` cannot be opened for reading.

```
reset_states(states=None)
```

tf.io.decode_and_crop_jpeg

- Contents
- Aliases:

Decode and Crop a JPEG-encoded image to a uint8 tensor.

Aliases:

- `tf.compat.v1.image.decode_and_crop_jpeg`
- `tf.compat.v1.io.decode_and_crop_jpeg`
- `tf.compat.v2.image.decode_and_crop_jpeg`
- `tf.compat.v2.io.decode_and_crop_jpeg`
- `tf.image.decode_and_crop_jpeg`
- `tf.io.decode_and_crop_jpeg`

```
tf.io.decode_and_crop_jpeg(

    contents,

    crop_window,

    channels=0,
```

```
    ratio=1,

    fancy_upscaling=True,

    try_recover_truncated=False,

    acceptable_fraction=1,

    dct_method='',

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
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.
It is equivalent to a combination of decode and crop, but much faster by only decoding partial jpeg image.

Args:

- **`contents`**: A `Tensor` of type `string`. 0-D. The JPEG-encoded image.
- **`crop_window`**: A `Tensor` of type `int32`. 1-D. The crop window: [crop_y, crop_x, crop_height, crop_width].
- **`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.
- **`dct_method`**: An optional `string`. Defaults to `""`. string specifying a hint about the algorithm used for decompression. Defaults to "" which maps to a system-specific default. Currently valid values are ["INTEGER_FAST", "INTEGER_ACCURATE"]. The hint may be ignored (e.g., the internal jpeg library changes to a version that does not have that specific option.)
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `uint8`.

tf.io.decode_base64

- Contents

- Aliases:

Decode web-safe base64-encoded strings.

Aliases:

- `tf.compat.v1.decode_base64`
- `tf.compat.v1.io.decode_base64`
- `tf.compat.v2.io.decode_base64`
- `tf.io.decode_base64`

```
tf.io.decode_base64(

    input,

    name=None

)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Input may or may not have padding at the end. See EncodeBase64 for padding. Web-safe means that input must use - and _ instead of + and /.

Args:

- **`input`**: A `Tensor` of type `string`. Base64 strings to decode.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.io.decode_bmp

- Contents
- Aliases:

Decode the first frame of a BMP-encoded image to a uint8 tensor.

Aliases:

- `tf.compat.v1.image.decode_bmp`
- `tf.compat.v1.io.decode_bmp`
- `tf.compat.v2.image.decode_bmp`
- `tf.compat.v2.io.decode_bmp`
- `tf.image.decode_bmp`
- `tf.io.decode_bmp`

```
tf.io.decode_bmp(

    contents,

    channels=0,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
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 BMP-encoded image.
- 3: output an RGB image.
- 4: output an RGBA image.

Args:

- **`contents`**: A `Tensor` of type `string`. 0-D. The BMP-encoded image.
- **`channels`**: An optional `int`. Defaults to `0`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `uint8`.

tf.io.decode_compressed

- Contents
- Aliases:

Decompress strings.

Aliases:

- `tf.compat.v1.decode_compressed`
- `tf.compat.v1.io.decode_compressed`
- `tf.compat.v2.io.decode_compressed`
- `tf.io.decode_compressed`

```
tf.io.decode_compressed(

    bytes,

    compression_type='',

    name=None

)
```

Defined in generated file: `python/ops/gen_parsing_ops.py`.
This op decompresses each element of the `bytes` input `Tensor`, which is assumed to be compressed using the given `compression_type`.
The `output` is a string `Tensor` of the same shape as `bytes`, each element containing the decompressed data from the corresponding element in `bytes`.

Args:

- **`bytes`**: A `Tensor` of type `string`. A Tensor of string which is compressed.
- **`compression_type`**: An optional `string`. Defaults to `""`. A scalar containing either (i) the empty string (no compression), (ii) "ZLIB", or (iii) "GZIP".
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `string`.

tf.io.decode_csv

- Contents

- Aliases:

Convert CSV records to tensors. Each column maps to one tensor.

Aliases:

- `tf.compat.v2.io.decode_csv`
- `tf.io.decode_csv`

```
tf.io.decode_csv(

    records,

    record_defaults,

    field_delim=',',

    use_quote_delim=True,

    na_value='',

    select_cols=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

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 specific types. Acceptable types are `float32`, `float64`, `int32`, `int64`, `string`. One tensor per column of the input record, with either a scalar default value for that column or an empty vector if the column is required.
- **`field_delim`**: An optional `string`. Defaults to `","`. char delimiter to separate fields in a record.
- **`use_quote_delim`**: An optional `bool`. Defaults to `True`. If false, treats double quotation marks as regular characters inside of the string fields (ignoring RFC 4180, Section 2, Bullet 5).
- **`na_value`**: Additional string to recognize as NA/NaN.
- **`select_cols`**: Optional sorted list of column indices to select. If specified, only this subset of columns will be parsed and returned.
- **`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.

Raises:

- **`ValueError`**: If any of the arguments is malformed.

tf.io.decode_gif

- Contents

- Aliases:

Decode the frame(s) of a GIF-encoded image to a uint8 tensor.

Aliases:

- `tf.compat.v1.image.decode_gif`
- `tf.compat.v1.io.decode_gif`
- `tf.compat.v2.image.decode_gif`
- `tf.compat.v2.io.decode_gif`
- `tf.image.decode_gif`
- `tf.io.decode_gif`

```
tf.io.decode_gif(

    contents,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.

GIF images with frame or transparency compression are not supported. On Linux and MacOS systems, convert animated GIFs from compressed to uncompressed by running:

```
convert $src.gif -coalesce $dst.gif
```

This op also supports decoding JPEGs and PNGs, though it is cleaner to use`tf.image.decode_image`.

Args:

- **`contents`**: A `Tensor` of type `string`. 0-D. The GIF-encoded image.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `uint8`.

tf.io.decode_image

- Contents
- Aliases:
- Used in the tutorials:

Function for `decode_bmp`, `decode_gif`, `decode_jpeg`, and `decode_png`.

Aliases:

- `tf.compat.v1.image.decode_image`
- `tf.compat.v1.io.decode_image`
- `tf.compat.v2.image.decode_image`
- `tf.compat.v2.io.decode_image`
- `tf.image.decode_image`
- `tf.io.decode_image`

```
tf.io.decode_image(
```

```
    contents,
    channels=None,
    dtype=tf.dtypes.uint8,
    name=None,
    expand_animations=True
)
```

Defined in `python/ops/image_ops_impl.py`.

Used in the tutorials:

- Load images with tf.data
- Neural style transfer

Detects whether an image is a BMP, GIF, JPEG, or PNG, and performs the appropriate operation to convert the input bytes `string` into a `Tensor` of type `dtype`.

Note: `decode_gif` returns a 4-D array `[num_frames, height, width, 3]`, as opposed to `decode_bmp`, `decode_jpeg` and `decode_png`, which return 3-D arrays `[height, width, num_channels]`. Make sure to take this into account when constructing your graph if you are intermixing GIF files with BMP, JPEG, and/or PNG files. Alternately, set the `expand_animations` argument of this function to `False`, in which case the op will return 3-dimensional tensors and will truncate animated GIF files to the first frame.

Args:

- `contents`: 0-D `string`. The encoded image bytes.
- `channels`: An optional `int`. Defaults to `0`. Number of color channels for the decoded image.
- `dtype`: The desired DType of the returned `Tensor`.
- `name`: A name for the operation (optional)
- `expand_animations`: Controls the shape of the returned op's output. If `True`, the returned op will produce a 3-D tensor for PNG, JPEG, and BMP files; and a 4-D tensor for all GIFs, whether animated or not. If, `False`, the returned op will produce a 3-D tensor for all file types and will truncate animated GIFs to the first frame.

Returns:

`Tensor` with type `dtype` and a 3- or 4-dimensional shape, depending on the file type and the value of the `expand_animations` parameter.

Raises:

- `ValueError`: On incorrect number of channels.

tf.io.decode_jpeg

- Contents
- Aliases:
- Used in the tutorials:

Decode a JPEG-encoded image to a uint8 tensor.

Aliases:

- `tf.compat.v1.image.decode_jpeg`

- `tf.compat.v1.io.decode_jpeg`
- `tf.compat.v2.image.decode_jpeg`
- `tf.compat.v2.io.decode_jpeg`
- `tf.image.decode_jpeg`
- `tf.io.decode_jpeg`

```
tf.io.decode_jpeg(

    contents,

    channels=0,

    ratio=1,

    fancy_upscaling=True,

    try_recover_truncated=False,

    acceptable_fraction=1,

    dct_method='',

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.

Used in the tutorials:

- [Image Captioning with Attention]
- [Load images with tf.data]
- [Pix2Pix]
- [Using TFRecords and tf.Example]

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.

This op also supports decoding PNGs and non-animated GIFs since the interface is the same, though it is cleaner to use `tf.image.decode_image`.

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.
- `dct_method`: An optional `string`. Defaults to `""`. string specifying a hint about the algorithm used for decompression. Defaults to "" which maps to a system-specific default. Currently valid values are ["INTEGER_FAST", "INTEGER_ACCURATE"]. The hint may be ignored (e.g., the internal jpeg library changes to a version that does not have that specific option.)
- `name`: A name for the operation (optional).

Returns:
A `Tensor` of type `uint8`.

tf.io.decode_json_example

- Contents
- Aliases:

Convert JSON-encoded Example records to binary protocol buffer strings.

Aliases:

- `tf.compat.v1.decode_json_example`
- `tf.compat.v1.io.decode_json_example`
- `tf.compat.v2.io.decode_json_example`
- `tf.io.decode_json_example`

```
tf.io.decode_json_example(

    json_examples,

    name=None

)
```

Defined in generated file: `python/ops/gen_parsing_ops.py`.

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`.

tf.io.decode_png

- Contents
- Aliases:

Decode a PNG-encoded image to a uint8 or uint16 tensor.

Aliases:

- `tf.compat.v1.image.decode_png`
- `tf.compat.v1.io.decode_png`

- `tf.compat.v2.image.decode_png`
- `tf.compat.v2.io.decode_png`
- `tf.image.decode_png`
- `tf.io.decode_png`

```
tf.io.decode_png(

    contents,

    channels=0,

    dtype=tf.dtypes.uint8,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
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.
This op also supports decoding JPEGs and non-animated GIFs since the interface is the same, though it is cleaner to use `tf.image.decode_image`.

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`.

tf.io.decode_proto

- Contents
- Aliases:

The op extracts fields from a serialized protocol buffers message into tensors.

Aliases:

- `tf.compat.v1.io.decode_proto`
- `tf.compat.v2.io.decode_proto`
- `tf.io.decode_proto`

```
tf.io.decode_proto(

    bytes,
```

```
    message_type,

    field_names,

    output_types,

    descriptor_source='local://',

    message_format='binary',

    sanitize=False,

    name=None

)
```

Defined in generated file: `python/ops/gen_decode_proto_ops.py`.
The `decode_proto` op extracts fields from a serialized protocol buffers message into tensors. The fields in `field_names` are decoded and converted to the corresponding `output_types` if possible.
A `message_type` name must be provided to give context for the field names. The actual message descriptor can be looked up either in the linked-in descriptor pool or a filename provided by the caller using the `descriptor_source` attribute.
Each output tensor is a dense tensor. This means that it is padded to hold the largest number of repeated elements seen in the input minibatch. (The shape is also padded by one to prevent zero-sized dimensions). The actual repeat counts for each example in the minibatch can be found in the `sizes`output. In many cases the output of `decode_proto` is fed immediately into tf.squeeze if missing values are not a concern. When using tf.squeeze, always pass the squeeze dimension explicitly to avoid surprises.
For the most part, the mapping between Proto field types and TensorFlow dtypes is straightforward. However, there are a few special cases:

- A proto field that contains a submessage or group can only be converted to `DT_STRING` (the serialized submessage). This is to reduce the complexity of the API. The resulting string can be used as input to another instance of the decode_proto op.
- TensorFlow lacks support for unsigned integers. The ops represent uint64 types as a `DT_INT64`with the same twos-complement bit pattern (the obvious way). Unsigned int32 values can be represented exactly by specifying type `DT_INT64`, or using twos-complement if the caller specifies `DT_INT32` in the `output_types` attribute.

The `descriptor_source` attribute selects a source of protocol descriptors to consult when looking up `message_type`. This may be a filename containing a serialized `FileDescriptorSet` message, or the special value `local://`, in which case only descriptors linked into the code will be searched; the filename can be on any filesystem accessible to TensorFlow.
You can build a `descriptor_source` file using the `--descriptor_set_out` and `--include_imports` options to the protocol compiler `protoc`.
The `local://` database only covers descriptors linked into the code via C++ libraries, not Python imports. You can link in a proto descriptor by creating a cc_library target with alwayslink=1.
Both binary and text proto serializations are supported, and can be chosen using the `format` attribute.

Args:

- **`bytes`**: A `Tensor` of type `string`. Tensor of serialized protos with shape `batch_shape`.

- **message_type**: A `string`. Name of the proto message type to decode.
- **field_names**: A list of `strings`. List of strings containing proto field names. An extension field can be decoded by using its full name, e.g. EXT_PACKAGE.EXT_FIELD_NAME.
- **output_types**: A list of `tf.DTypes`. List of TF types to use for the respective field in field_names.
- **descriptor_source**: An optional `string`. Defaults to `"local://"`. Either the special value `local://` or a path to a file containing a serialized `FileDescriptorSet`.
- **message_format**: An optional `string`. Defaults to `"binary"`. Either `binary` or `text`.
- **sanitize**: An optional `bool`. Defaults to `False`. Whether to sanitize the result or not.
- **name**: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (sizes, values).

- **sizes**: A `Tensor` of type `int32`.
- **values**: A list of `Tensor` objects of type `output_types`.

tf.io.decode_raw

- Contents
- Aliases:

Convert raw byte strings into tensors.

Aliases:

- `tf.compat.v2.io.decode_raw`
- `tf.io.decode_raw`

```
tf.io.decode_raw(

    input_bytes,

    out_type,

    little_endian=True,

    fixed_length=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

Args:

- **input_bytes**: Each element of the input Tensor is converted to an array of bytes.
- **out_type**: `DType` of the output. Acceptable types are `half`, `float`, `double`, `int32`, `uint16`, `uint8`, `int16`, `int8`, `int64`.
- **little_endian**: Whether the `input_bytes` data is in little-endian format. Data will be converted into host byte order if necessary.
- **fixed_length**: If set, the first `fixed_length` bytes of each element will be converted. Data will be zero-padded or truncated to the specified length. `fixed_length` must be a multiple of the size of `out_type`. `fixed_length` must be specified if the elements of `input_bytes` are of variable length.
- **name**: A name for the operation (optional).

Returns:
A `Tensor` object storing the decoded bytes.

tf.io.deserialize_many_sparse

- Contents
- Aliases:

Deserialize and concatenate `SparseTensors` from a serialized minibatch.

Aliases:

- `tf.compat.v1.deserialize_many_sparse`
- `tf.compat.v1.io.deserialize_many_sparse`
- `tf.compat.v2.io.deserialize_many_sparse`
- `tf.io.deserialize_many_sparse`

```
tf.io.deserialize_many_sparse(

    serialized_sparse,

    dtype,

    rank=None,

    name=None

)
```

Defined in `python/ops/sparse_ops.py`.
The input `serialized_sparse` must be a string matrix of shape `[N x 3]` where `N` is the minibatch size and the rows correspond to packed outputs of `serialize_sparse`. The ranks of the original `SparseTensor` objects must all match. When the final `SparseTensor` is created, it has rank one higher than the ranks of the incoming `SparseTensor` objects (they have been concatenated along a new row dimension).
The output `SparseTensor` object's shape values for all dimensions but the first are the max across the input `SparseTensor` objects' shape values for the corresponding dimensions. Its first shape value is `N`, the minibatch size.
The input `SparseTensor` objects' indices are assumed ordered in standard lexicographic order. If this is not the case, after this step run `sparse.reorder` to restore index ordering.
For example, if the serialized input is a `[2, 3]` matrix representing two original `SparseTensor`objects:

```
index = [ 0]

        [10]

        [20]

values = [1, 2, 3]

shape = [50]
```

and

```
index = [ 2]

        [10]

values = [4, 5]

shape = [30]
```

then the final deserialized `SparseTensor` will be:

```
index = [0  0]

        [0 10]

        [0 20]

        [1  2]

        [1 10]

values = [1, 2, 3, 4, 5]

shape = [2 50]
```

Args:

- **`serialized_sparse`**: 2-D `Tensor` of type `string` of shape `[N, 3]`. The serialized and packed `SparseTensor` objects.
- **`dtype`**: The `dtype` of the serialized `SparseTensor` objects.
- **`rank`**: (optional) Python int, the rank of the `SparseTensor` objects.
- **`name`**: A name prefix for the returned tensors (optional)

Returns:

A `SparseTensor` representing the deserialized `SparseTensor`s, concatenated along the `SparseTensor`s' first dimension.

All of the serialized `SparseTensor`s must have had the same rank and type.

tf.io.encode_base64

- Contents
- Aliases:

Encode strings into web-safe base64 format.

Aliases:

- `tf.compat.v1.encode_base64`
- `tf.compat.v1.io.encode_base64`
- `tf.compat.v2.io.encode_base64`
- `tf.io.encode_base64`

```
tf.io.encode_base64(
    input,
    pad=False,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.
Refer to the following article for more information on base64 format: en.wikipedia.org/wiki/Base64.
Base64 strings may have padding with '=' at the end so that the encoded has length multiple of 4.
See Padding section of the link above.
Web-safe means that the encoder uses - and _ instead of + and /.

Args:

- **`input`**: A `Tensor` of type `string`. Strings to be encoded.
- **`pad`**: An optional `bool`. Defaults to `False`. Bool whether padding is applied at the ends.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.io.encode_jpeg

- Contents
- Aliases:

JPEG-encode an image.

Aliases:

- `tf.compat.v1.image.encode_jpeg`
- `tf.compat.v1.io.encode_jpeg`
- `tf.compat.v2.image.encode_jpeg`
- `tf.compat.v2.io.encode_jpeg`
- `tf.image.encode_jpeg`
- `tf.io.encode_jpeg`

```
tf.io.encode_jpeg(
    image,
    format='',
    quality=95,
    progressive=False,
    optimize_size=False,
    chroma_downsampling=True,
```

```
    density_unit='in',

    x_density=300,

    y_density=300,

    xmp_metadata='',

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
`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`. See http://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`.

tf.io.encode_proto

- Contents
- Aliases:

The op serializes protobuf messages provided in the input tensors.

Aliases:

- `tf.compat.v1.io.encode_proto`
- `tf.compat.v2.io.encode_proto`
- `tf.io.encode_proto`

```
tf.io.encode_proto(

    sizes,

    values,

    field_names,

    message_type,

    descriptor_source='local://',

    name=None

)
```

Defined in generated file: `python/ops/gen_encode_proto_ops.py`.

The types of the tensors in `values` must match the schema for the fields specified in `field_names`. All the tensors in `values` must have a common shape prefix, *batch_shape*.

The `sizes` tensor specifies repeat counts for each field. The repeat count (last dimension) of a each tensor in `values` must be greater than or equal to corresponding repeat count in `sizes`.

A `message_type` name must be provided to give context for the field names. The actual message descriptor can be looked up either in the linked-in descriptor pool or a filename provided by the caller using the `descriptor_source` attribute.

The `descriptor_source` attribute selects a source of protocol descriptors to consult when looking up `message_type`. This may be a filename containing a serialized `FileDescriptorSet` message, or the special value `local://`, in which case only descriptors linked into the code will be searched; the filename can be on any filesystem accessible to TensorFlow.

You can build a `descriptor_source` file using the `--descriptor_set_out` and `--include_imports` options to the protocol compiler `protoc`.

The `local://` database only covers descriptors linked into the code via C++ libraries, not Python imports. You can link in a proto descriptor by creating a cc_library target with alwayslink=1.

There are a few special cases in the value mapping:

Submessage and group fields must be pre-serialized as TensorFlow strings.

TensorFlow lacks support for unsigned int64s, so they must be represented as `tf.int64` with the same twos-complement bit pattern (the obvious way).

Unsigned int32 values can be represented exactly with `tf.int64`, or with sign wrapping if the input is of type `tf.int32`.

Args:

- **`sizes`**: A `Tensor` of type `int32`. Tensor of int32 with shape `[batch_shape, len(field_names)]`.
- **`values`**: A list of `Tensor` objects. List of tensors containing values for the corresponding field.
- **`field_names`**: A list of `strings`. List of strings containing proto field names.
- **`message_type`**: A `string`. Name of the proto message type to decode.
- **`descriptor_source`**: An optional `string`. Defaults to `"local://"`.

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

Returns:
A `Tensor` of type `string`.

tf.io.extract_jpeg_shape

- Contents
- Aliases:

Extract the shape information of a JPEG-encoded image.

Aliases:

- `tf.compat.v1.image.extract_jpeg_shape`
- `tf.compat.v1.io.extract_jpeg_shape`
- `tf.compat.v2.image.extract_jpeg_shape`
- `tf.compat.v2.io.extract_jpeg_shape`
- `tf.image.extract_jpeg_shape`
- `tf.io.extract_jpeg_shape`

```
tf.io.extract_jpeg_shape(

    contents,

    output_type=tf.dtypes.int32,

    name=None

)
```

Defined in generated file: `python/ops/gen_image_ops.py`.
This op only parses the image header, so it is much faster than DecodeJpeg.

Args:

- **`contents`**: A `Tensor` of type `string`. 0-D. The JPEG-encoded image.
- **`output_type`**: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int32`. (Optional) The output type of the operation (int32 or int64). Defaults to int32.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `output_type`.

tf.io.FixedLenFeature

- Contents
- Class FixedLenFeature
  - Aliases:
  - Used in the tutorials:
- Properties

Class `FixedLenFeature`

Configuration for parsing a fixed-length input feature.

Aliases:

- Class `tf.compat.v1.FixedLenFeature`
- Class `tf.compat.v1.io.FixedLenFeature`

- Class `tf.compat.v2.io.FixedLenFeature`
- Class `tf.io.FixedLenFeature`

Defined in `python/ops/parsing_ops.py`.

Used in the tutorials:

- Using TFRecords and tf.Example

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` and of the specified `shape`.

Properties

`shape`

`dtype`

`default_value`

tf.io.FixedLenSequenceFeature

- Contents
- Class FixedLenSequenceFeature
  - Aliases:
- Properties
  - shape

Class `FixedLenSequenceFeature`

Configuration for parsing a variable-length input feature into a `Tensor`.

Aliases:

- Class `tf.compat.v1.FixedLenSequenceFeature`
- Class `tf.compat.v1.io.FixedLenSequenceFeature`
- Class `tf.compat.v2.io.FixedLenSequenceFeature`
- Class `tf.io.FixedLenSequenceFeature`

Defined in `python/ops/parsing_ops.py`.

The resulting `Tensor` of parsing a single `SequenceExample` or `Example` has a static `shape` of `[None] + shape` and the specified `dtype`. The resulting `Tensor` of parsing a `batch_size` many `Example`s has a static `shape` of `[batch_size, None] + shape` and the specified `dtype`. The entries in the `batch` from different `Examples` will be padded with `default_value` to the maximum length present in the `batch`.

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 for dimension 2 and higher. First dimension is of variable length `None`.
- **`dtype`**: Data type of input.
- **`allow_missing`**: Whether to allow this feature to be missing from a feature list item. Is available only for parsing `SequenceExample` not for parsing `Examples`.

- **`default_value`**: Scalar value to be used to pad multiple `Example`s to their maximum length. Irrelevant for parsing a single `Example` or `SequenceExample`. Defaults to "" for dtype string and 0 otherwise (optional).

Properties

`shape`

`dtype`

`allow_missing`

`default_value`

tf.io.is_jpeg

- Contents
- Aliases:

Convenience function to check if the 'contents' encodes a JPEG image.

Aliases:

- `tf.compat.v1.image.is_jpeg`
- `tf.compat.v1.io.is_jpeg`
- `tf.compat.v2.image.is_jpeg`
- `tf.compat.v2.io.is_jpeg`
- `tf.image.is_jpeg`
- `tf.io.is_jpeg`

```
tf.io.is_jpeg(

    contents,

    name=None

)
```

Defined in `python/ops/image_ops_impl.py`.

Args:

- **`contents`**: 0-D `string`. The encoded image bytes.
- **`name`**: A name for the operation (optional)

Returns:

A scalar boolean tensor indicating if 'contents' may be a JPEG image. is_jpeg is susceptible to false positives.

tf.io.matching_files

- Contents
- Aliases:

Returns the set of files matching one or more glob patterns.

Aliases:

- `tf.compat.v1.io.matching_files`
- `tf.compat.v1.matching_files`
- `tf.compat.v2.io.matching_files`

- `tf.io.matching_files`

```
tf.io.matching_files(

    pattern,

    name=None

)
```

Defined in generated file: `python/ops/gen_io_ops.py`.

Note that this routine only supports wildcard characters in the basename portion of the pattern, not in the directory portion. Note also that the order of filenames returned can be non-deterministic.

Args:

- **`pattern`**: A `Tensor` of type `string`. Shell wildcard pattern(s). Scalar or vector of type string.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.io.match_filenames_once

- Contents
- Aliases:

Save the list of files matching pattern, so it is only computed once.

Aliases:

- `tf.compat.v1.io.match_filenames_once`
- `tf.compat.v1.train.match_filenames_once`
- `tf.compat.v2.io.match_filenames_once`
- `tf.io.match_filenames_once`

```
tf.io.match_filenames_once(

    pattern,

    name=None

)
```

Defined in `python/training/input.py`.

NOTE: The order of the files returned can be non-deterministic.

Args:

- **`pattern`**: A file pattern (glob), or 1D tensor of file patterns.
- **`name`**: A name for the operations (optional).

Returns:

A variable that is initialized to the list of files matching the pattern(s).

tf.io.parse_example

- Contents
- Aliases:

Parses `Example` protos into a `dict` of tensors.

Aliases:

- `tf.compat.v2.io.parse_example`
- `tf.io.parse_example`

```
tf.io.parse_example(

    serialized,

    features,

    example_names=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

Parses a number of serialized `Example` protos given in `serialized`. We refer to `serialized` as a batch with `batch_size` many entries of individual `Example` protos.

`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`, `SparseFeature`, and `FixedLenFeature`objects. Each `VarLenFeature` and `SparseFeature` 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` identifies the example in `serialized`, and `index` is the value's index in the list of values associated with that feature and example.

Each `SparseFeature` maps to a `SparseTensor` of the specified type representing a Tensor of `dense_shape [batch_size] + SparseFeature.size`. Its `values` come from the feature in the examples with key `value_key`. A `values[i]` comes from a position `k` in the feature of an example at batch entry `batch`. This positional information is recorded in `indices[i]` as `[batch, index_0, index_1, ...]` where `index_j` is the `k-th` value of the feature in the example at with key `SparseFeature.index_key[j]`. In other words, we split the indices (except the first index indicating the batch entry) of a `SparseTensor` by dimension into different features of the `Example`. Due to its complexity a `VarLenFeature` should be preferred over a `SparseFeature` whenever possible.

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`.

Each `FixedLenSequenceFeature` `df` maps to a `Tensor` of the specified type (or `tf.float32` if not specified) and shape `(serialized.size(), None) + df.shape`. All examples in `serialized` will be padded with `default_value` along the second dimension.

Examples:
For example, if one expects a `tf.float32` `VarLenFeature` `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],

                    dense_shape=(3, 2)) }
```

If instead a `FixedLenSequenceFeature` with `default_value = -1.0` and `shape=[]` is used then the output will look like:

```
{"ft": [[1.0, 2.0], [3.0, -1.0]]}
```

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.float32),

}
```

Then the output is a dictionary:

```
{

  "kw": SparseTensor(

      indices=[[0, 0], [0, 1], [1, 0]],

      values=["knit", "big", "emmy"]

      dense_shape=[2, 2]),

  "dank": SparseTensor(

      indices=[[1, 0]],

      values=[42],

      dense_shape=[2, 1]),

  "gps": SparseTensor(

      indices=[],

      values=[],
```

```
      dense_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"]],

}
```

An alternative to `VarLenFeature` to obtain a `SparseTensor` is `SparseFeature`. For example, given two `Example` input protos in `serialized`:

```
[
  features {
    feature { key: "val" value { float_list { value: [ 0.5, -1.0 ] } } }
    feature { key: "ix" value { int64_list { value: [ 3, 20 ] } } }
  },
  features {
    feature { key: "val" value { float_list { value: [ 0.0 ] } } }
    feature { key: "ix" value { int64_list { value: [ 42 ] } } }
  }
]
```

And arguments

```
example_names: ["input0", "input1"],
features: {
    "sparse": SparseFeature(
        index_key="ix", value_key="val", dtype=tf.float32, size=100),
}
```

Then the output is a dictionary:

```
{
  "sparse": SparseTensor(
      indices=[[0, 3], [0, 20], [1, 42]],
      values=[0.5, -1.0, 0.0]
      dense_shape=[2, 100]),
```

```
}
```

Args:

- **`serialized`**: A vector (1-D Tensor) of strings, a batch of binary serialized `Example` protos.
- **`features`**: A `dict` mapping feature keys to `FixedLenFeature`, `VarLenFeature`, and `SparseFeature` values.
- **`example_names`**: A vector (1-D Tensor) of strings (optional), the names of the serialized protos in the batch.
- **`name`**: A name for this operation (optional).

Returns:

A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:

- **`ValueError`**: if any feature is invalid.

tf.io.parse_sequence_example

- Contents
- Aliases:

Parses a batch of `SequenceExample` protos.

Aliases:

- `tf.compat.v1.io.parse_sequence_example`
- `tf.compat.v2.io.parse_sequence_example`
- `tf.io.parse_sequence_example`

```
tf.io.parse_sequence_example(

    serialized,

    context_features=None,

    sequence_features=None,

    example_names=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

Parses a vector of serialized `SequenceExample` protos given in `serialized`.

This op parses serialized sequence examples into a tuple of dictionaries mapping keys to `Tensor` and `SparseTensor` objects respectively. The first dictionary contains mappings for keys appearing in`context_features`, and the second dictionary contains mappings for keys appearing in `sequence_features`.

At least one of `context_features` and `sequence_features` must be provided and non-empty.

The `context_features` keys are associated with a `SequenceExample` as a whole, independent of time / frame. In contrast, the `sequence_features` keys provide a way to access variable-length data within the `FeatureList` section of the `SequenceExample` proto. While the shapes

of `context_features` values are fixed with respect to frame, the frame dimension (the first dimension) of `sequence_features` values may vary between `SequenceExample` protos, and even between `feature_list` keys within the same `SequenceExample`.
`context_features` contains `VarLenFeature` and `FixedLenFeature` objects.
Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenFeature` is mapped to a `Tensor`, of the specified type, shape, and default value.
`sequence_features` contains `VarLenFeature` and `FixedLenSequenceFeature` objects.
Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenSequenceFeature` is mapped to a `Tensor`, each of the specified type. The shape will be `(B,T,) + df.dense_shape` for `FixedLenSequenceFeature` `df`, where `B` is the batch size, and `T` is the length of the associated`FeatureList` in the `SequenceExample`. For instance, `FixedLenSequenceFeature([])` yields a scalar 2-D `Tensor` of static shape `[None, None]` and dynamic shape `[B, T]`, while`FixedLenSequenceFeature([k])` (for `int k >= 1`) yields a 3-D matrix `Tensor` of static shape `[None, None, k]` and dynamic shape `[B, T, k]`.
Like the input, the resulting output tensors have a batch dimension. This means that the original per-example shapes of `VarLenFeature`s and `FixedLenSequenceFeature`s can be lost. To handle that situation, this op also provides dicts of shape tensors as part of the output. There is one dict for the context features, and one for the feature_list features. Context features of type `FixedLenFeature`s will not be present, since their shapes are already known by the caller. In situations where the input 'FixedLenFeature`s are of different lengths across examples, the shorter examples will be padded with default datatype values: 0 for numeric types, and the empty string for string types.
Each `SparseTensor` corresponding to `sequence_features` represents a ragged vector. Its indices are `[time, index]`, where `time` is the `FeatureList` entry and `index` is the value's index in the list of values associated with that time.
`FixedLenFeature` entries with a `default_value` and `FixedLenSequenceFeature` entries with `allow_missing=True` are optional; otherwise, we will fail if that `Feature` or `FeatureList` is missing from any example in `serialized`.
`example_name` may contain a descriptive name for the corresponding serialized proto. This may be useful for debugging purposes, but it has no effect on the output. If not `None`, `example_name` must be a scalar.

Args:

- **`serialized`**: A vector (1-D Tensor) of type string containing binary serialized `SequenceExample` protos.
- **`context_features`**: A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature`values. These features are associated with a `SequenceExample` as a whole.
- **`sequence_features`**: A `dict` mapping feature keys to `FixedLenSequenceFeature` or `VarLenFeature` values. These features are associated with data within the `FeatureList`section of the `SequenceExample` proto.
- **`example_names`**: A vector (1-D Tensor) of strings (optional), the name of the serialized protos.
- **`name`**: A name for this operation (optional).

Returns:

A tuple of three `dict`s, each mapping keys to `Tensor`s and `SparseTensor`s. The first dict contains the context key/values, the second dict contains the feature_list key/values, and the final dict contains the lengths of any dense feature_list features.

Raises:

- **`ValueError`**: if any feature is invalid.

tf.io.parse_single_example

- Contents
- Aliases:

- Used in the tutorials:
  Parses a single `Example` proto.

  Aliases:
- `tf.compat.v2.io.parse_single_example`
- `tf.io.parse_single_example`

```
tf.io.parse_single_example(

    serialized,

    features,

    example_names=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

Used in the tutorials:
- Using TFRecords and tf.Example
  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).
  One might see performance advantages by batching `Example` protos with `parse_example` instead of using this function directly.

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.
- **`example_names`**: (Optional) A scalar string Tensor, the associated name.
  See `_parse_single_example_raw` documentation for more details.
- **`name`**: A name for this operation (optional).

Returns:
A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:
- **`ValueError`**: if any feature is invalid.

tf.io.parse_single_sequence_example

- Contents
- Aliases:
  Parses a single `SequenceExample` proto.

  Aliases:
- `tf.compat.v1.io.parse_single_sequence_example`

- `tf.compat.v1.parse_single_sequence_example`
- `tf.compat.v2.io.parse_single_sequence_example`
- `tf.io.parse_single_sequence_example`

```
tf.io.parse_single_sequence_example(

    serialized,

    context_features=None,

    sequence_features=None,

    example_name=None,

    name=None

)
```

Defined in `python/ops/parsing_ops.py`.

Parses a single serialized `SequenceExample` proto given in `serialized`.

This op parses a serialized sequence example into a tuple of dictionaries mapping keys to `Tensor`and `SparseTensor` objects respectively. The first dictionary contains mappings for keys appearing in`context_features`, and the second dictionary contains mappings for keys appearing in `sequence_features`.

At least one of `context_features` and `sequence_features` must be provided and non-empty.

The `context_features` keys are associated with a `SequenceExample` as a whole, independent of time / frame. In contrast, the `sequence_features` keys provide a way to access variable-length data within the `FeatureList` section of the `SequenceExample` proto. While the shapes of `context_features` values are fixed with respect to frame, the frame dimension (the first dimension) of `sequence_features` values may vary between `SequenceExample` protos, and even between `feature_list` keys within the same `SequenceExample`.

`context_features` contains `VarLenFeature` and `FixedLenFeature` objects. Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenFeature` is mapped to a `Tensor`, of the specified type, shape, and default value.

`sequence_features` contains `VarLenFeature` and `FixedLenSequenceFeature` objects. Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenSequenceFeature` is mapped to a `Tensor`, each of the specified type. The shape will be `(T,) + df.dense_shape` for `FixedLenSequenceFeature` `df`, where `T` is the length of the associated `FeatureList` in the `SequenceExample`. For instance, `FixedLenSequenceFeature([])` yields a scalar 1-D `Tensor` of static shape `[None]` and dynamic shape `[T]`, while `FixedLenSequenceFeature([k])` (for `int k >= 1`) yields a 2-D matrix `Tensor` of static shape `[None, k]` and dynamic shape `[T, k]`.

Each `SparseTensor` corresponding to `sequence_features` represents a ragged vector. Its indices are `[time, index]`, where `time` is the `FeatureList` entry and `index` is the value's index in the list of values associated with that time.

`FixedLenFeature` entries with a `default_value` and `FixedLenSequenceFeature` entries with `allow_missing=True` are optional; otherwise, we will fail if that `Feature` or `FeatureList` is missing from any example in `serialized`.

`example_name` may contain a descriptive name for the corresponding serialized proto. This may be useful for debugging purposes, but it has no effect on the output. If not `None`, `example_name` must be a scalar.

Args:

- **`serialized`**: A scalar (0-D Tensor) of type string, a single binary serialized `SequenceExample`proto.
- **`context_features`**: A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature`values. These features are associated with a `SequenceExample` as a whole.
- **`sequence_features`**: A `dict` mapping feature keys to `FixedLenSequenceFeature` or `VarLenFeature` values. These features are associated with data within the `FeatureList`section of the `SequenceExample` proto.
- **`example_name`**: A scalar (0-D Tensor) of strings (optional), the name of the serialized proto.
- **`name`**: A name for this operation (optional).

Returns:

A tuple of two `dict`s, each mapping keys to `Tensor`s and `SparseTensor`s. The first dict contains the context key/values. The second dict contains the feature_list key/values.

Raises:

- **`ValueError`**: if any feature is invalid.

tf.io.parse_tensor

- Contents
- Aliases:
- Used in the tutorials:

Transforms a serialized tensorflow.TensorProto proto into a Tensor.

Aliases:

- `tf.compat.v1.io.parse_tensor`
- `tf.compat.v1.parse_tensor`
- `tf.compat.v2.io.parse_tensor`
- `tf.io.parse_tensor`

```
tf.io.parse_tensor(

    serialized,

    out_type,

    name=None

)
```

Defined in generated file: `python/ops/gen_parsing_ops.py`.

Used in the tutorials:

- Load images with tf.data

Args:

- **`serialized`**: A `Tensor` of type `string`. A scalar string containing a serialized TensorProto proto.
- **`out_type`**: A `tf.DType`. The type of the serialized tensor. The provided type must match the type of the serialized tensor and no implicit conversion will take place.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `out_type`.

tf.io.read_file

- Contents
- Aliases:
- Used in the tutorials:

Reads and outputs the entire contents of the input filename.

Aliases:

- `tf.compat.v1.io.read_file`
- `tf.compat.v1.read_file`
- `tf.compat.v2.io.read_file`
- `tf.io.read_file`

```
tf.io.read_file(

    filename,

    name=None

)
```

Defined in generated file: `python/ops/gen_io_ops.py`.

Used in the tutorials:

- Image Captioning with Attention
- Load images with tf.data
- Neural style transfer
- Pix2Pix

Args:

- **`filename`**: A `Tensor` of type `string`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `string`.

tf.io.serialize_many_sparse

- Contents
- Aliases:

Serialize `N`-minibatch `SparseTensor` into an `[N, 3]` `Tensor`.

Aliases:

- `tf.compat.v2.io.serialize_many_sparse`
- `tf.io.serialize_many_sparse`

```
tf.io.serialize_many_sparse(

    sp_input,

    out_type=tf.dtypes.string,
```

```
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

The `SparseTensor` must have rank `R` greater than 1, and the first dimension is treated as the minibatch dimension. Elements of the `SparseTensor` must be sorted in increasing order of this first dimension. The serialized `SparseTensor` objects going into each row of the output `Tensor` will have rank `R-1`.

The minibatch size `N` is extracted from `sparse_shape[0]`.

Args:

- **`sp_input`**: The input rank `R` `SparseTensor`.
- **`out_type`**: The `dtype` to use for serialization.
- **`name`**: A name prefix for the returned tensors (optional).

Returns:

A matrix (2-D `Tensor`) with `N` rows and `3` columns. Each column represents serialized `SparseTensor`'s indices, values, and shape (respectively).

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.io.serialize_sparse

- Contents
- Aliases:

Serialize a `SparseTensor` into a 3-vector (1-D `Tensor`) object.

Aliases:

- `tf.compat.v2.io.serialize_sparse`
- `tf.io.serialize_sparse`

```
tf.io.serialize_sparse(
    sp_input,
    out_type=tf.dtypes.string,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

Args:

- **`sp_input`**: The input `SparseTensor`.
- **`out_type`**: The `dtype` to use for serialization.
- **`name`**: A name prefix for the returned tensors (optional).

Returns:

A 3-vector (1-D `Tensor`), with each column representing the serialized `SparseTensor`'s indices, values, and shape (respectively).

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.io.serialize_tensor

- Contents
- Aliases:

Transforms a Tensor into a serialized TensorProto proto.

Aliases:

- `tf.compat.v1.io.serialize_tensor`
- `tf.compat.v1.serialize_tensor`
- `tf.compat.v2.io.serialize_tensor`
- `tf.io.serialize_tensor`

```
tf.io.serialize_tensor(

    tensor,

    name=None

)
```

Defined in generated file: `python/ops/gen_parsing_ops.py`.

Args:

- **`tensor`**: A `Tensor`. A Tensor of type `T`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.io.SparseFeature

- Contents
- Class SparseFeature
  - Aliases:
- Properties
  - index_key

Class `SparseFeature`

Configuration for parsing a sparse input feature from an `Example`.

Aliases:

- Class `tf.compat.v1.SparseFeature`
- Class `tf.compat.v1.io.SparseFeature`
- Class `tf.compat.v2.io.SparseFeature`
- Class `tf.io.SparseFeature`

Defined in `python/ops/parsing_ops.py`.

Note, preferably use `VarLenFeature` (possibly in combination with a `SequenceExample`) in order to parse out `SparseTensor`s instead of `SparseFeature` due to its simplicity.

Closely mimicking the `SparseTensor` that will be obtained by parsing an `Example` with a `SparseFeature` config, a `SparseFeature` contains a

- `value_key`: The name of key for a `Feature` in the `Example` whose parsed `Tensor` will be the resulting `SparseTensor.values`.
- `index_key`: A list of names - one for each dimension in the resulting `SparseTensor` whose `indices[i][dim]` indicating the position of the `i`-th value in the `dim` dimension will be equal to the `i`-th value in the Feature with key named `index_key[dim]` in the `Example`.
- `size`: A list of ints for the resulting `SparseTensor.dense_shape`.

For example, we can represent the following 2D `SparseTensor`

```
SparseTensor(indices=[[3, 1], [20, 0]],

             values=[0.5, -1.0]

             dense_shape=[100, 3])
```

with an `Example` input proto

```
features {

  feature { key: "val" value { float_list { value: [ 0.5, -1.0 ] } } }

  feature { key: "ix0" value { int64_list { value: [ 3, 20 ] } } }

  feature { key: "ix1" value { int64_list { value: [ 1, 0 ] } } }

}
```

and `SparseFeature` config with 2 `index_key`s

```
SparseFeature(index_key=["ix0", "ix1"],

              value_key="val",

              dtype=tf.float32,

              size=[100, 3])
```

Fields:

- **`index_key`**: A single string name or a list of string names of index features. For each key the underlying feature's type must be `int64` and its length must always match that of the `value_key` feature. To represent `SparseTensor`s with a `dense_shape` of `rank` higher than 1 a list of length `rank` should be used.
- **`value_key`**: Name of value feature. The underlying feature's type must be `dtype` and its length must always match that of all the `index_key`s' features.
- **`dtype`**: Data type of the `value_key` feature.
- **`size`**: A Python int or list thereof specifying the dense shape. Should be a list if and only if `index_key` is a list. In that case the list must be equal to the length of `index_key`. Each for each entry `i` all values in the `index_key`[i] feature must be in `[0, size[i])`.

- **`already_sorted`**: A Python boolean to specify whether the values in `value_key` are already sorted by their index position. If so skip sorting. False by default (optional).

Properties

`index_key`

`value_key`

`dtype`

`size`

`already_sorted`

tf.io.TFRecordOptions

- Contents

Class `TFRecordOptions`

Options used for manipulating TFRecord files.

Aliases:

- Class `tf.compat.v1.io.TFRecordOptions`
- Class `tf.compat.v1.python_io.TFRecordOptions`
- Class `tf.compat.v2.io.TFRecordOptions`
- Class `tf.io.TFRecordOptions`

Defined in `python/lib/io/tf_record.py`.

`__init__`

```
__init__(

    compression_type=None,

    flush_mode=None,

    input_buffer_size=None,

    output_buffer_size=None,

    window_bits=None,

    compression_level=None,

    compression_method=None,
```

```
    mem_level=None,

    compression_strategy=None

)
```

Creates a `TFRecordOptions` instance.
Options only effect TFRecordWriter when compression_type is not `None`. Documentation, details, and defaults can be found in `zlib_compression_options.h` and in the zlib manual. Leaving an option as `None` allows C++ to set a reasonable default.

Args:

- **`compression_type`**: `"GZIP"`, `"ZLIB"`, or `""` (no compression).
- **`flush_mode`**: flush mode or `None`, Default: Z_NO_FLUSH.
- **`input_buffer_size`**: int or `None`.
- **`output_buffer_size`**: int or `None`.
- **`window_bits`**: int or `None`.
- **`compression_level`**: 0 to 9, or `None`.
- **`compression_method`**: compression method or `None`.
- **`mem_level`**: 1 to 9, or `None`.
- **`compression_strategy`**: strategy or `None`. Default: Z_DEFAULT_STRATEGY.

Returns:
A `TFRecordOptions` object.

Raises:

- **`ValueError`**: If compression_type is invalid.

Methods

get_compression_type_string

```
@classmethod

get_compression_type_string(

    cls,

    options

)
```

Convert various option types to a unified string.

Args:

- **`options`**: `TFRecordOption`, `TFRecordCompressionType`, or string.

Returns:
Compression type as string (e.g. `'ZLIB'`, `'GZIP'`, or `''`).

Raises:

- **`ValueError`**: If compression_type is invalid.

Class Members

- `compression_type_map`

tf.io.TFRecordWriter

Class `TFRecordWriter`

A class to write records to a TFRecords file.

Aliases:

- Class `tf.compat.v1.io.TFRecordWriter`
- Class `tf.compat.v1.python_io.TFRecordWriter`
- Class `tf.compat.v2.io.TFRecordWriter`
- Class `tf.io.TFRecordWriter`

Defined in `python/lib/io/tf_record.py`.

Used in the tutorials:

- Using TFRecords and tf.Example

This class implements `__enter__` and `__exit__`, and can be used in `with` blocks like a normal file.

`__init__`

```
__init__(

    path,

    options=None

)
```

Opens file `path` and creates a `TFRecordWriter` writing to it.

Args:

- **`path`**: The path to the TFRecords file.
- **`options`**: (optional) String specifying compression type, `TFRecordCompressionType`, or `TFRecordOptions` object.

Raises:

- **`IOError`**: If `path` cannot be opened for writing.
- **`ValueError`**: If valid compression_type can't be determined from `options`.

Methods

`__enter__`

```
__enter__()
```

Enter a `with` block.

__exit__

```
__exit__(

    unused_type,

    unused_value,

    unused_traceback

)
```

Exit a `with` block, closing the file.

close

```
close()
```

Close the file.

flush

```
flush()
```

Flush the file.

write

```
write(record)
```

Write a string record to the file.

Args:

- **`record`**: str

tf.io.VarLenFeature

- Contents
- Class VarLenFeature
  - Aliases:
- Properties
  - dtype

Class `VarLenFeature`

Configuration for parsing a variable-length input feature.

Aliases:

- Class `tf.compat.v1.VarLenFeature`
- Class `tf.compat.v1.io.VarLenFeature`
- Class `tf.compat.v2.io.VarLenFeature`
- Class `tf.io.VarLenFeature`

Defined in `python/ops/parsing_ops.py`.

Fields:

- **`dtype`**: Data type of input.

Properties

`dtype`

tf.io.write_file

- Contents
- Aliases:

Writes contents to the file at input filename. Creates file and recursively

Aliases:

- `tf.compat.v1.io.write_file`
- `tf.compat.v1.write_file`
- `tf.compat.v2.io.write_file`
- `tf.io.write_file`

```
tf.io.write_file(

    filename,

    contents,

    name=None

)
```

Defined in generated file: `python/ops/gen_io_ops.py`.
creates directory if not existing.

Args:

- **`filename`**: A `Tensor` of type `string`. scalar. The name of the file to which we write the contents.
- **`contents`**: A `Tensor` of type `string`. scalar. The content to be written to the output file.
- **`name`**: A name for the operation (optional).

Returns:
The created Operation.

tf.io.write_graph

- Contents
- Aliases:

Writes a graph proto to a file.

Aliases:

- `tf.compat.v1.io.write_graph`
- `tf.compat.v1.train.write_graph`
- `tf.compat.v2.io.write_graph`
- `tf.io.write_graph`

```
tf.io.write_graph(
```

```
    graph_or_graph_def,

    logdir,

    name,

    as_text=True

)
```

Defined in `python/framework/graph_io.py`.
The graph is written as a text proto unless `as_text` is `False`.

```
v = tf.Variable(0, name='my_variable')

sess = tf.compat.v1.Session()

tf.io.write_graph(sess.graph_def, '/tmp/my-model', 'train.pbtxt')
```

or

```
v = tf.Variable(0, name='my_variable')

sess = tf.compat.v1.Session()

tf.io.write_graph(sess.graph, '/tmp/my-model', 'train.pbtxt')
```

Args:

- **`graph_or_graph_def`**: A `Graph` or a `GraphDef` protocol buffer.
- **`logdir`**: Directory where to write the graph. This can refer to remote filesystems, such as Google Cloud Storage (GCS).
- **`name`**: Filename for the graph.
- **`as_text`**: If `True`, writes the graph as an ASCII proto.

Returns:
The path of the output proto file.

Module: tf.io.gfile

- **Contents**
- Classes
- Functions

Public API for tf.io.gfile namespace.

Classes

`class GFile`: File I/O wrappers without thread locking.

Functions

`copy(...)`: Copies data from `src` to `dst`.
`exists(...)`: Determines whether a path exists or not.

`glob(...)`: Returns a list of files that match the given pattern(s).
`isdir(...)`: Returns whether the path is a directory or not.
`listdir(...)`: Returns a list of entries contained within a directory.
`makedirs(...)`: Creates a directory and all parent/intermediate directories.
`mkdir(...)`: Creates a directory with the name given by 'path'.
`remove(...)`: Deletes the path located at 'path'.
`rename(...)`: Rename or move a file / directory.
`rmtree(...)`: Deletes everything under path recursively.
`stat(...)`: Returns file statistics for a given path.
`walk(...)`: Recursive directory tree generator for directories.

Module: tf.nest

Public API for tf.nest namespace.

Functions

`assert_same_structure(...)`: Asserts that two structures are nested in the same way.
`flatten(...)`: Returns a flat list from a given nested structure.
`is_nested(...)`: Returns true if its input is a collections.Sequence (except strings).
`map_structure(...)`: Applies `func` to each entry in `structure` and returns a new structure.
`pack_sequence_as(...)`: Returns a given flattened sequence packed into a given structure.

tf.nest.assert_same_structure

Asserts that two structures are nested in the same way.

Aliases:

- `tf.compat.v1.nest.assert_same_structure`
- `tf.compat.v2.nest.assert_same_structure`
- `tf.nest.assert_same_structure`

```
tf.nest.assert_same_structure(

    nest1,

    nest2,

    check_types=True,

    expand_composites=False

)
```

Defined in `python/util/nest.py`.

Note that namedtuples with identical name and fields are always considered to have the same shallow structure (even with `check_types=True`). For instance, this code will print `True`:

```
def nt(a, b):
```

```
  return collections.namedtuple('foo', 'a b')(a, b)

print(assert_same_structure(nt(0, 1), nt(2, 3)))
```

Args:

- **`nest1`**: an arbitrarily nested structure.
- **`nest2`**: an arbitrarily nested structure.
- **`check_types`**: if `True` (default) types of sequences are checked as well, including the keys of dictionaries. If set to `False`, for example a list and a tuple of objects will look the same if they have the same size. Note that namedtuples with identical name and fields are always considered to have the same shallow structure. Two types will also be considered the same if they are both list subtypes (which allows "list" and "_ListWrapper" from trackable dependency tracking to compare equal).
- **`expand_composites`**: If true, then composite tensors such as `tf.SparseTensor` and `tf.RaggedTensor` are expanded into their component tensors.

Raises:

- **`ValueError`**: If the two structures do not have the same number of elements or if the two structures are not nested in the same way.
- **`TypeError`**: If the two structures differ in the type of sequence in any of their substructures. Only possible if `check_types` is `True`.

tf.nest.flatten

- Contents
- Aliases:

Returns a flat list from a given nested structure.

Aliases:

- `tf.compat.v1.nest.flatten`
- `tf.compat.v2.nest.flatten`
- `tf.nest.flatten`

```
tf.nest.flatten(

    structure,

    expand_composites=False

)
```

Defined in `python/util/nest.py`.

If nest is not a sequence, tuple, or dict, then returns a single-element list: [nest].

In the case of dict instances, the sequence consists of the values, sorted by key to ensure deterministic behavior. This is true also for OrderedDict instances: their sequence order is ignored, the sorting order of keys is used instead. The same convention is followed in pack_sequence_as. This correctly repacks dicts and OrderedDicts after they have been flattened, and also allows flattening an OrderedDict and then repacking it back using a corresponding plain dict, or vice-versa. Dictionaries with non-sortable keys cannot be flattened.

Users must not modify any collections used in nest while this function is running.

Args:

- **`structure`**: an arbitrarily nested structure or a scalar object. Note, numpy arrays are considered scalars.
- **`expand_composites`**: If true, then composite tensors such as tf.SparseTensor and tf.RaggedTensor are expanded into their component tensors.

Returns:

A Python list, the flattened version of the input.

Raises:

- **`TypeError`**: The nest is or contains a dict with non-sortable keys.

tf.nest.is_nested

- Contents
- Aliases:

Returns true if its input is a collections.Sequence (except strings).

Aliases:

- `tf.compat.v1.nest.is_nested`
- `tf.compat.v2.nest.is_nested`
- `tf.nest.is_nested`

```
tf.nest.is_nested(seq)
```

Defined in `python/util/nest.py`.

Args:

- **`seq`**: an input sequence.

Returns:

True if the sequence is a not a string and is a collections.Sequence or a dict.

tf.nest.map_structure

- Contents
- Aliases:

Applies `func` to each entry in `structure` and returns a new structure.

Aliases:

- `tf.compat.v1.nest.map_structure`
- `tf.compat.v2.nest.map_structure`
- `tf.nest.map_structure`

```
tf.nest.map_structure(

    func,

    *structure,

    **kwargs

)
```

Defined in `python/util/nest.py`.

Applies `func(x[0], x[1], ...)` where x[i] is an entry in `structure[i]`. All structures in `structure` must have the same arity, and the return value will contain results with the same structure layout.

Args:

- **`func`**: A callable that accepts as many arguments as there are structures.
- **`*structure`**: scalar, or tuple or list of constructed scalars and/or other tuples/lists, or scalars. Note: numpy arrays are considered as scalars.
- **`**kwargs`**: Valid keyword args are:
- `check_types`: If set to `True` (default) the types of iterables within the structures have to be same (e.g. `map_structure(func, [1], (1,))` raises a `TypeError` exception). To allow this set this argument to `False`. Note that namedtuples with identical name and fields are always considered to have the same shallow structure.
- `expand_composites`: If set to `True`, then composite tensors such as `tf.SparseTensor`and `tf.RaggedTensor` are expanded into their component tensors. If `False` (the default), then composite tensors are not expanded.

Returns:

A new structure with the same arity as `structure`, whose values correspond to `func(x[0], x[1], ...)` where `x[i]` is a value in the corresponding location in `structure[i]`. If there are different sequence types and `check_types` is `False` the sequence types of the first structure will be used.

Raises:

- **`TypeError`**: If `func` is not callable or if the structures do not match each other by depth tree.
- **`ValueError`**: If no structure is provided or if the structures do not match each other by type.
- **`ValueError`**: If wrong keyword arguments are provided.

tf.nest.pack_sequence_as

- Contents
- Aliases:

Returns a given flattened sequence packed into a given structure.

Aliases:

- `tf.compat.v1.nest.pack_sequence_as`
- `tf.compat.v2.nest.pack_sequence_as`
- `tf.nest.pack_sequence_as`

```
tf.nest.pack_sequence_as(

    structure,

    flat_sequence,

    expand_composites=False

)
```

Defined in `python/util/nest.py`.

If `structure` is a scalar, `flat_sequence` must be a single-element list; in this case the return value is `flat_sequence[0]`.

If `structure` is or contains a dict instance, the keys will be sorted to pack the flat sequence in deterministic order. This is true also for `OrderedDict` instances: their sequence order is ignored, the sorting order of keys is used instead. The same convention is followed in `flatten`. This correctly repacks dicts and `OrderedDict`s after they have been flattened, and also allows flattening an `OrderedDict` and then repacking it back using a corresponding plain dict, or vice-versa. Dictionaries with non-sortable keys cannot be flattened.

Args:

- **`structure`**: Nested structure, whose structure is given by nested lists, tuples, and dicts. Note: numpy arrays and strings are considered scalars.
- **`flat_sequence`**: flat sequence to pack.
- **`expand_composites`**: If true, then composite tensors such as `tf.SparseTensor` and `tf.RaggedTensor` are expanded into their component tensors.

Returns:

- **`packed`**: `flat_sequence` converted to have the same recursive structure as `structure`.

Raises:

- **`ValueError`**: If `flat_sequence` and `structure` have different element counts.
- **`TypeError`**: `structure` is or contains a dict with non-sortable keys.