|  |
| --- |
|  |

I am improving my answer to add more details for saving and restoring models.

In(and after) Tensorflow version 0.11:

Save the model:

import tensorflow as tf

#Prepare to feed input, i.e. feed\_dict and placeholders

w1 = tf.placeholder("float", name="w1")

w2 = tf.placeholder("float", name="w2")

b1= tf.Variable(2.0,name="bias")

feed\_dict ={w1:4,w2:8}

#Define a test operation that we will restore

w3 = tf.add(w1,w2)

w4 = tf.multiply(w3,b1,name="op\_to\_restore")

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

#Create a saver object which will save all the variables

saver = tf.train.Saver()

#Run the operation by feeding input

print sess.run(w4,feed\_dict)

#Prints 24 which is sum of (w1+w2)\*b1

#Now, save the graph

saver.save(sess, 'my\_test\_model',global\_step=1000)

Restore the model:

import tensorflow as tf

sess=tf.Session()

#First let's load meta graph and restore weights

saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')

saver.restore(sess,tf.train.latest\_checkpoint('./'))

# Access saved Variables directly

print(sess.run('bias:0'))

# This will print 2, which is the value of bias that we saved

# Now, let's access and create placeholders variables and

# create feed-dict to feed new data

graph = tf.get\_default\_graph()

w1 = graph.get\_tensor\_by\_name("w1:0")

w2 = graph.get\_tensor\_by\_name("w2:0")

feed\_dict ={w1:13.0,w2:17.0}

#Now, access the op that you want to run.

op\_to\_restore = graph.get\_tensor\_by\_name("op\_to\_restore:0")

print sess.run(op\_to\_restore,feed\_dict)

#This will print 60 which is calculated

In( and After) TensorFlow version 0.11.0RC1, you can save and restore your model directly by calling tf.train.export\_meta\_graph and tf.train.import\_meta\_graph according to https://www.tensorflow.org/programmers\_guide/meta\_graph

save model:

w1 = tf.Variable(tf.truncated\_normal(shape=[10]), name='w1')

w2 = tf.Variable(tf.truncated\_normal(shape=[20]), name='w2')

tf.add\_to\_collection('vars', w1)

tf.add\_to\_collection('vars', w2)

saver = tf.train.Saver()

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

saver.save(sess, 'my-model')

# `save` method will call `export\_meta\_graph` implicitly.

# you will get saved graph files:my-model.meta

restore model:

sess = tf.Session()

new\_saver = tf.train.import\_meta\_graph('my-model.meta')

new\_saver.restore(sess, tf.train.latest\_checkpoint('./'))

all\_vars = tf.get\_collection('vars')

for v in all\_vars:

v\_ = sess.run(v)

print(v\_)

In this Tensorflow tutorial, I shall explain:

1. *How does a Tensorflow model look like?*
2. *How to save a Tensorflow model?*
3. *How to restore a Tensorflow model for prediction/transfer learning*?
4. How to work with imported pretrained models for fine-tuning and modification

This tutorial assumes that you have some idea about training a neural network. Otherwise, please [follow this tutorial](http://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/) and come back here.

**1.What is a Tensorflow model?:**

After you have trained a neural network, you would want to save it for future use and deploying to production. So, what is a Tensorflow model? Tensorflow model primarily contains the network design or graph and values of the network parameters that we have trained. Hence, Tensorflow model has two main files:

**a) Meta graph:**

This is a protocol buffer which saves the complete Tensorflow graph; i.e. all variables, operations, collections etc. This file has **.meta** extension.

**b) Checkpoint file:**

This is a binary file which contains all the values of the weights, biases, gradients and all the other variables saved. This file has an extension .**ckpt.** However, Tensorflow has changed this from version 0.11. Now, instead of single .ckpt file, we have two files:

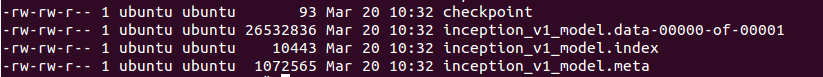
Python

|  |  |
| --- | --- |
| 1  2  3 | mymodel.data-00000-of-00001  mymodel.index |

.data file is the file that contains our training variables and we shall go after it.

Along with this, Tensorflow also has a file named **checkpoint** which simply keeps a record of latest checkpoint files saved.

So, to summarize, Tensorflow models for versions greater than 0.10 look like this:



while Tensorflow model before 0.11 contained only three files:

Python

|  |  |
| --- | --- |
| 1  2  3  4 | inception\_v1.meta  inception\_v1.ckpt  checkpoint |

Now that we know how a Tensorflow model looks like, let’s learn how to save the model.

**2. Saving a Tensorflow model:**

Let’s say, you are [training a convolutional neural network for image classification](http://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/). As a standard practice, you keep a watch on loss and accuracy numbers. Once you see that the network has converged, you can stop the training manually or you will run the training for fixed number of epochs. After the training is done, we want to save all the variables and network graph to a file for future use. So, in Tensorflow, you want to save the graph and values of all the parameters for which we shall be creating an instance of tf.train.Saver() class.

saver = tf.train.Saver()

Remember that Tensorflow variables are only alive inside a session. So, you have to save the model inside a session by calling save method on saver object you just created.

Python

|  |  |
| --- | --- |
| 1  2 | saver.save(sess, 'my-test-model') |

Here, sess is the session object, while ‘my-test-model’ is the name you want to give your model. Let’s see a complete example:

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import tensorflow as tf  w1 = tf.Variable(tf.random\_normal(shape=[2]), name='w1')  w2 = tf.Variable(tf.random\_normal(shape=[5]), name='w2')  saver = tf.train.Saver()  sess = tf.Session()  sess.run(tf.global\_variables\_initializer())  saver.save(sess, 'my\_test\_model')    # This will save following files in Tensorflow v >= 0.11  # my\_test\_model.data-00000-of-00001  # my\_test\_model.index  # my\_test\_model.meta  # checkpoint |

If we are saving the model after 1000 iterations, we shall call save by passing the step count:

saver.save(sess, 'my\_test\_model',global\_step=1000)

This will just append ‘-1000’ to the model name and following files will be created:

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | my\_test\_model-1000.index  my\_test\_model-1000.meta  my\_test\_model-1000.data-00000-of-00001  checkpoint |

Let’s say, while training, we are saving our model after every 1000 iterations, so .meta file is created the first time(on 1000th iteration) and we don’t need to recreate the .meta file each time(so, we don’t save the .meta file at 2000, 3000.. or any other iteration). We only save the model for further iterations, as the graph will not change. Hence, when we don’t want to write the meta-graph we use this:

Python

|  |  |
| --- | --- |
| 1  2 | saver.save(sess, 'my-model', global\_step=step,write\_meta\_graph=False) |

If you want to keep only 4 latest models and want to save one model after every 2 hours during training you can use max\_to\_keep and keep\_checkpoint\_every\_n\_hours like this.

Python

|  |  |
| --- | --- |
| 1  2  3 | #saves a model every 2 hours and maximum 4 latest models are saved.  saver = tf.train.Saver(max\_to\_keep=4, keep\_checkpoint\_every\_n\_hours=2) |

**Note, if we don’t specify anything in the tf.train.Saver(), it saves all the variables**. What if, we don’t want to save all the variables and just some of them. We can specify the variables/collections we want to save. While creating the tf.train.Saver instance we pass it a list or a dictionary of variables that we want to save. Let’s look at an example:

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import tensorflow as tf  w1 = tf.Variable(tf.random\_normal(shape=[2]), name='w1')  w2 = tf.Variable(tf.random\_normal(shape=[5]), name='w2')  saver = tf.train.Saver([w1,w2])  sess = tf.Session()  sess.run(tf.global\_variables\_initializer())  saver.save(sess, 'my\_test\_model',global\_step=1000) |

This can be used to save specific part of Tensorflow graphs when required.

**3. Importing a pre-trained model:**

If you want to use someone else’s pre-trained model for fine-tuning, there are two things you need to do:

**a) Create the network:**

You can create the network by writing python code to create each and every layer manually as the original model. However, if you think about it, we had saved the network in .meta file which we can use to recreate the network using tf.train.import() function like this: saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')

Remember, import\_meta\_graph appends the network defined in .meta file to the current graph. So, this will create the graph/network for you but we still need to load the value of the parameters that we had trained on this graph.

**b) Load the parameters:**

We can restore the parameters of the network by calling restore on this saver which is an instance of tf.train.Saver() class.

Python

|  |  |
| --- | --- |
| 1  2  3  4 | with tf.Session() as sess:    new\_saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')    new\_saver.restore(sess, tf.train.latest\_checkpoint('./')) |

After this, the value of tensors like w1 and w2 has been restored and can be accessed:

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | with tf.Session() as sess:      saver = tf.train.import\_meta\_graph('my-model-1000.meta')      saver.restore(sess,tf.train.latest\_checkpoint('./'))      print(sess.run('w1:0'))  ##Model has been restored. Above statement will print the saved value of w1. |

So, now you have understood how saving and importing works for a Tensorflow model. In the next section, I have described a practical usage of above to load any pre-trained model.

**4. Working with restored models**

Now that you have understood how to save and restore Tensorflow models, Let’s develop a practical guide to restore any pre-trained model and use it for prediction, fine-tuning or further training. Whenever you are working with Tensorflow, you define a graph which is fed examples(training data) and some hyperparameters like learning rate, global step etc. It’s a standard practice to feed all the training data and hyperparameters using placeholders. Let’s build a small network using placeholders and save it. Note that when the network is saved, values of the placeholders are not saved.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import tensorflow as tf    #Prepare to feed input, i.e. feed\_dict and placeholders  w1 = tf.placeholder("float", name="w1")  w2 = tf.placeholder("float", name="w2")  b1= tf.Variable(2.0,name="bias")  feed\_dict ={w1:4,w2:8}    #Define a test operation that we will restore  w3 = tf.add(w1,w2)  w4 = tf.multiply(w3,b1,name="op\_to\_restore")  sess = tf.Session()  sess.run(tf.global\_variables\_initializer())    #Create a saver object which will save all the variables  saver = tf.train.Saver()    #Run the operation by feeding input  print sess.run(w4,feed\_dict)  #Prints 24 which is sum of (w1+w2)\*b1    #Now, save the graph  saver.save(sess, 'my\_test\_model',global\_step=1000) |

Now, when we want to restore it, we not only have to restore the graph and weights, but also prepare a new feed\_dict that will feed the new training data to the network. We can get reference to these saved operations and placeholder variables via **graph.get\_tensor\_by\_name()** method.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #How to access saved variable/Tensor/placeholders  w1 = graph.get\_tensor\_by\_name("w1:0")    ## How to access saved operation  op\_to\_restore = graph.get\_tensor\_by\_name("op\_to\_restore:0") |

If we just want to run the same network with different data, you can simply pass the new data via feed\_dict to the network.

Restoring model and retraining with your own data

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import tensorflow as tf    sess=tf.Session()  #First let's load meta graph and restore weights  saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')  saver.restore(sess,tf.train.latest\_checkpoint('./'))      # Now, let's access and create placeholders variables and  # create feed-dict to feed new data    graph = tf.get\_default\_graph()  w1 = graph.get\_tensor\_by\_name("w1:0")  w2 = graph.get\_tensor\_by\_name("w2:0")  feed\_dict ={w1:13.0,w2:17.0}    #Now, access the op that you want to run.  op\_to\_restore = graph.get\_tensor\_by\_name("op\_to\_restore:0")    print sess.run(op\_to\_restore,feed\_dict)  #This will print 60 which is calculated  #using new values of w1 and w2 and saved value of b1. |

What if you want to add more operations to the graph by adding more layers and then train it. Of course you can do that too. See here:

Restoring model and retraining with your own data

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import tensorflow as tf    sess=tf.Session()  #First let's load meta graph and restore weights  saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')  saver.restore(sess,tf.train.latest\_checkpoint('./'))      # Now, let's access and create placeholders variables and  # create feed-dict to feed new data    graph = tf.get\_default\_graph()  w1 = graph.get\_tensor\_by\_name("w1:0")  w2 = graph.get\_tensor\_by\_name("w2:0")  feed\_dict ={w1:13.0,w2:17.0}    #Now, access the op that you want to run.  op\_to\_restore = graph.get\_tensor\_by\_name("op\_to\_restore:0")    #Add more to the current graph  add\_on\_op = tf.multiply(op\_to\_restore,2)    print sess.run(add\_on\_op,feed\_dict)  #This will print 120. |

But, can you restore part of the old graph and add-on to that for fine-tuning ? Of-course you can, just access the appropriate operation by graph.get\_tensor\_by\_name() method and build graph on top of that. Here is a real world example. Here we load a vgg pre-trained network using meta graph and change the number of outputs to 2 in the last layer for fine-tuning with new data.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | ......  ......  saver = tf.train.import\_meta\_graph('vgg.meta')  # Access the graph  graph = tf.get\_default\_graph()  ## Prepare the feed\_dict for feeding data for fine-tuning    #Access the appropriate output for fine-tuning  fc7= graph.get\_tensor\_by\_name('fc7:0')    #use this if you only want to change gradients of the last layer  fc7 = tf.stop\_gradient(fc7) # It's an identity function  fc7\_shape= fc7.get\_shape().as\_list()    new\_outputs=2  weights = tf.Variable(tf.truncated\_normal([fc7\_shape[3], num\_outputs], stddev=0.05))  biases = tf.Variable(tf.constant(0.05, shape=[num\_outputs]))  output = tf.matmul(fc7, weights) + biases  pred = tf.nn.softmax(output)    # Now, you run this with fine-tuning data in sess.run() |

Hopefully, this gives you very clear understanding of how Tensorflow models are saved and restored. Please feel free to share your questions or doubts in the comments section.

[TensorFlow](http://cv-tricks.com/tag/tensorflow/), [Tensorflow tutorial](http://cv-tricks.com/tag/tensorflow-tutorial/)

arrow\_upwardClass Saver

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

See the guides: Exporting and Importing a MetaGraph > Exporting a Complete Model to MetaGraph, Exporting and Importing a MetaGraph, Variables > Saving and Restoring Variables

Saves and restores variables.

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

The Saver class adds ops to save and restore variables to and from checkpoints. It also provides convenience methods to run these ops.

Checkpoints are binary files in a proprietary format which map variable names to tensor values. The best way to examine the contents of a checkpoint is to load it using a Saver.

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

You number checkpoint filenames by passing a value to the optional global\_step argument to save():

hdr\_strong

content\_copy

saver.save(sess, 'my-model', global\_step=0) ==> filename: 'my-model-0'

...

saver.save(sess, 'my-model', global\_step=1000) ==> filename: 'my-model-1000'

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

•max\_to\_keep indicates the maximum number of recent checkpoint files to keep. As new files are created, older files are deleted. If None or 0, all checkpoint files are kept. Defaults to 5 (that is, the 5 most recent checkpoint files are kept.)

•keep\_checkpoint\_every\_n\_hours: In addition to keeping the most recent max\_to\_keep checkpoint files, you might want to keep one checkpoint file for every N hours of training. This can be useful if you want to later analyze how a model progressed during a long training session. For example, passing keep\_checkpoint\_every\_n\_hours=2 ensures that you keep one checkpoint file for every 2 hours of training. The default value of 10,000 hours effectively disables the feature.

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

A training program that saves regularly looks like:

hdr\_strong

content\_copy

...

# Create a saver.

saver = tf.train.Saver(...variables...)

# Launch the graph and train, saving the model every 1,000 steps.

sess = tf.Session()

for step in xrange(1000000):

sess.run(..training\_op..)

if step % 1000 == 0:

# Append the step number to the checkpoint name:

saver.save(sess, 'my-model', global\_step=step)

In addition to checkpoint files, savers keep a protocol buffer on disk with the list of recent checkpoints. This is used to manage numbered checkpoint files and by latest\_checkpoint(), which makes it easy to discover the path to the most recent checkpoint. That protocol buffer is stored in a file named 'checkpoint' next to the checkpoint files.

If you create several savers, you can specify a different filename for the protocol buffer file in the call to save().

arrow\_upwardProperties

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.

arrow\_upwardMethods

\_\_init\_\_

hdr\_strong

content\_copy

\_\_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:

hdr\_strong

content\_copy

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

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

# Pass the variables as a dict:

saver = tf.train.Saver({'v1': v1, 'v2': v2})

# Or pass them as a list.

saver = tf.train.Saver([v1, v2])

# Passing a list is equivalent to passing a dict with the variable op names

# as keys:

saver = tf.train.Saver({v.op.name: v for v in [v1, v2]})

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

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

Args:

•var\_list: A list of Variable/SaveableObject, or a dictionary mapping names to SaveableObjects. 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 BaseSaverBuilder().

•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.

as\_saver\_def

hdr\_strong

content\_copy

as\_saver\_def()

Generates a SaverDef representation of this saver.

Returns:

A SaverDef proto.

build

hdr\_strong

content\_copy

build()

Builds saver\_def.

export\_meta\_graph

hdr\_strong

content\_copy

export\_meta\_graph(

filename=None,

collection\_list=None,

as\_text=False,

export\_scope=None,

clear\_devices=False,

clear\_extraneous\_savers=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.

Returns:

A MetaGraphDef proto.

from\_proto

hdr\_strong

content\_copy

from\_proto(

saver\_def,

import\_scope=None

)

Returns a Saver object created from saver\_def.

Args:

•saver\_def: a SaveDef protocol buffer.

•import\_scope: Optional string. Name scope to use.

Returns:

A Saver built from saver\_def.

recover\_last\_checkpoints

hdr\_strong

content\_copy

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

hdr\_strong

content\_copy

restore(

sess,

save\_path

)

Restores previously saved variables.

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

The save\_path argument is typically a value previously returned from a save() call, or a call to latest\_checkpoint().

Args:

•sess: A Session to use to restore the parameters.

•save\_path: Path where parameters were previously saved.

Raises:

•ValueError: If save\_path is None.

save

hdr\_strong

content\_copy

save(

sess,

save\_path,

global\_step=None,

latest\_filename=None,

meta\_graph\_suffix='meta',

write\_meta\_graph=True,

write\_state=True

)

Saves variables.

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

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

Args:

•sess: A Session to use to save the variables.

•save\_path: String. Path to the checkpoint filename. If the saver is sharded, this is the prefix of the sharded checkpoint filename.

•global\_step: If provided the global step number is appended to save\_path to create the checkpoint filename. The optional argument can be a Tensor, a Tensor name or an integer.

•latest\_filename: Optional name for the protocol buffer file that will contains the list of most recent checkpoint filenames. That file, kept in the same directory as the checkpoint files, is automatically managed by the saver to keep track of recent checkpoints. Defaults to 'checkpoint'.

•meta\_graph\_suffix: Suffix for MetaGraphDef file. Defaults to 'meta'.

•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.

Returns:

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

hdr\_strong

content\_copy

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

hdr\_strong

content\_copy

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

hdr\_strong

content\_copy

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.