Saving and Loading a TensorFlow model using the SavedModel API

[Go to the profile of Jose Flores](https://medium.com/@jsflo.dev?source=post_header_lockup)

[Jose Flores](https://medium.com/@jsflo.dev)Follow

Jun 7, 2018

The SavedModel API allows you to save a trained model into a format that can be easily loaded in Python, Java, *(soon JavaScrip*t*)*, upload to GCP: ML Engine or use a TensorFlow Serving server.

This post will cover saving a trained model in python and then loading that model in Java and Python.

**What’s Saved ?**

assets/  
assets.extra/  
**variables/**  
 **variables.data-\*\*\*\*\*-of-\*\*\*\*\***  
  **variables.index**  
**saved\_model.pb**

The **.pb** is the MetaGraphDef which holds the graph structure. The **variables** folder holds your **learned weights.**The **assets** folder allow you to add **external files** that may be needed and **assets.extra** is a place libraries can add their assets.

[***“MetaGraph****is a dataflow graph, plus its associated variables, assets, and signatures. A***MetaGraphDef***is the protocol buffer representation of a MetaGraph.”*](https://www.tensorflow.org/programmers_guide/saved_model)

**How is this different from**[**tf.train.Saver**](https://www.tensorflow.org/api_docs/python/tf/train/Saver)

This is different from the **Saver** API**(tf.train.Saver)** save method which only saves the **variables**by adding save and restore operations to the graph.

**# construct graph!**  
...  
**# add save/restore ops**  
saver = tf.train.Saver()  
...  
**# save after training**  
save\_path = saver.save(sess, "/tmp/**model.ckpt**")

The **Saver**API saves the **variables** in **checkpoint** files and **requires** you to **reconstruct**the graph in order to load the variables. This is desired when you are splitting your training into separate sessions and want a quick way to resume training. But for loading a model in a different language or to “package” a complete model the **SavedModel** API is recommended.

**Saving**

For this example we’ll use the [MNIST beginner tutorial used in the official TensorFlow documentation](https://www.tensorflow.org/versions/r1.0/get_started/mnist/beginners). We will modify the script in two ways: **adding names to ops** and adding a couple of lines to **save the model** after training.

**Adding names**

We’re adding names to the input and output operations so that we can reference the operations by name when we load it. This step isn’t really necessary but it does make it a lot easier to load.

x = tf.placeholder(tf.float32, [None, 784], **name="myInput"**)

Most, if not all, TF operations allow you to specify a name. For this example I was able to add the name to the **input placeholder.**

y = tf.nn.softmax(tf.matmul(x, W) + b, **name="myOutput"**)

Adding a name to the **output** for this script is also straight forward.

There are cases where you may have a final output node/tensor but didn’t get the chance to add a name for whatever reason. In those cases you could use [tf.identity](https://www.tensorflow.org/api_docs/python/tf/identity) which allows you to add a name given a Tensor.

def addNameToTensor(someTensor, theName):  
 return **tf.identity**(someTensor, **name=theName**)

**Save**

Add these lines after training using the same session (sess).

The easiest way to save is using the [tf.saved\_model.simple\_save](https://www.tensorflow.org/api_docs/python/tf/saved_model/simple_save) function:

simple\_save(sess,  
 export\_dir,  
 inputs={**"myInput"**: **x**},  
 outputs={**"myOutput"**: **y**})

**simple\_save** allows you to quickly save with the minimal amount of arguments needed. It uses some sensible defaults to provide this convenience and one of the most important, for our case, is the **tag**that is used.

The tag is used to distinguish different **MetaGraphDef** saved and is needed when loading the model. You can use any string for a tag but by default it uses tag\_constants.SERVING ([“serve”](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/tag_constants.py)). This also supports the Predict API which means any TensorFlow Serving server can load the model.

The way to actually save multiple **MetaGraphDef's**, to define your own tags or to include assets is to use the builder.

[*import tensorflow.python.saved\_model*](https://www.tensorflow.org/api_docs/python/tf/saved_model/builder/SavedModelBuilder) *from* tensorflow.python.saved\_model *import* tag\_constants  
*from tensorflow.python.saved\_model.signature\_def\_utils\_impl import predict\_signature\_def*

**builder = saved\_model.builder.SavedModelBuilder(export\_path)**  
  
**signature = predict\_signature\_def(inputs={'myInput': x},  
 outputs={'myOutput': y})**

# using custom tag instead of: tags=[tag\_constants.SERVING]  
**builder.add\_meta\_graph\_and\_variables(sess=sess,  
 tags=["myTag"],  
 signature\_def\_map={'predict': signature})  
builder.save()**

**Loading**

Loading is similar for Python and Java and will probably be the same for any language that supports that API. We need the directory of the Saved Model artifacts, the tag used and the names of the input/output tensors.

[The TensorFlow JavaScript API doesn’t support this format yet :(.](https://js.tensorflow.org/tutorials/model-save-load.html)

TensorFlow Serving and GCP have different independent steps but both were painless to use for my simple use cases, except for learning a bit about gRPC for TF Serving, but we won’t be covering these.

**Loading in Python**

**with tf.Session(graph=tf.Graph()) as sess:  
 tf.saved\_model.loader.load(sess, ["serve"], export\_path)**

*graph = tf.get\_default\_graph()  
 print(graph.get\_operations())*

**sess.run('myOutput:0',** feed\_dict=**{'myInput:0': *...***

After calling the **load** function the graph is loaded as the default graph so you can interact with the graph as if you had reconstructed the graph. The variables are also loaded so you can start running inference on any new data.

**Loading in Java**

*compile "org.tensorflow:tensorflow:1.8.0"*

*import org.tensorflow.\**

**SavedModelBundle savedModelBundle = SavedModelBundle.load("./export\_path", "serve");**

*Graph graph = savedModelBundle.graph();  
printOperations(graph);*

**Tensor result = savedModelBundle.session().runner()  
 .feed("myInput", tensorInput)  
 .fetch("myOutput")  
 .run().get(0);**

The TensorFlow Java Api hates boxed types so make sure you’re using primitives and it’s probably a good idea to validate your inputs **after**converting to Tensors if you’re getting wrong results.