TensorFlow: saving/restoring and mixing multiple models

[](https://blog.metaflow.fr/@morgangiraud?source=post_header_lockup)

[Morgan](https://blog.metaflow.fr/@morgangiraud)Follow

Nov 15, 2016

Before going any further, make sure you read the [very small primer I made on TF here](https://medium.com/@morgangiraud/tensorflow-a-primer-4b3fa0978be3#.6axq0wkd7)

Why start with that information? Because it is of tremendous importance to understand what can be saved at the different level of your code to avoid messing around cluelessly…

**How to actually save and load something**

**The Saver and Session object**

Any interaction with your filesystem to save persistent data in TF needs a[**Saver** object](https://www.tensorflow.org/api_docs/python/tf/train/Saver) and a [**Session** object](https://www.tensorflow.org/api_docs/python/tf/Session).

The **Saver** constructor allows you to control many things among which 1 is important:

* The **var\_list:** Default to None, this is the list of variables you want to persist to your filesystem. You can either choose to save all the variables, some variables or even a dictionary to give custom names to your variables.

The **Session** constructor allows you to control 3 things:

* The var\_list: This is used in case of a distributed architecture to handle computation. You can specify which TF server or ‘target’ you want to compute on.
* The **graph:**the graph you want the **Session**to handle. The tricky thing for beginners is the fact that there is always a **default Graph**in TF where all operations are set by default, so you are always in a “**default Graph scope**”.
* The **config**: You can use [**ConfigProto**](https://github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/core/protobuf/config.proto)to configure TF. Check the linked source for more details.

The **Saver**can handle the saving and loading (called restoring) of your **Graph**metadata and your **Variables** data. To do that, it adds operations inside the current **Graph**that will be evaluated within a session**.**

By default, the **Saver**will handle the **default** **Graph**and all its included **Variables,**but you can create as much **Savers**as you want to control any graph or subgraph and their variables.

Here is an example:

Basic example on how to use a Saver

If you look at your folder, it actually creates 3 files per save call and a checkpoint file, I’ll go into more details about this in the annexe. You can go on just by understanding that weights are saved into **.data**files and your graph and metadata are saved into the **.meta**file.

*Note: You must be careful to use a****Saver****with a****Session****linked to the****Graph****containing all the****variables****the Saver is handling.*😨

*Here is an example of how NOT to save a variable:*

Basic example on how NOT to use a Saver

**Restoring operations and other metadata**

One important information is the fact that the Saver will save any metadata associated with your **Graph**. It means that loading a .meta checkpoint will also restore all empty variables, operations and collections associated with your **Graph**(for example, it will restore the optimiser and its learning rate).

When you restore a meta checkpoint, you actually load your saved graph in the current default graph. Now you can access itto load anything inside like a tensor, an operation or a collection.

To restore a meta checkpoint, use the TF helper import\_meta\_graph:

**Restoring the weights**

Remember that actual real weights only exists within a **Session.**It means that the “restore” action must have access to a session to restore weights inside a **Graph.** The best way to understand the restore operation is to see it simply as a kind of initialisation.

**Using a pre-trained graph in a new graph**

Now that you know how to save and load, you can probably figure out how to do it. Yet, there might be some tricks that could help you go faster.

* Can the output of one graph be the input of another graph?

Yes, but there is a drawback to this: I don’t know yet a way to make the gradient flow easily between graphs, as you will have to evaluate the first graph, get the results and feed it to the next graph.

This can be ok until you need to retrain the first graph too. In that case, you will need to grab the inputs gradients to feed it to the training step of your first graph…

* Can I mix all of those different graphs in only one graph?

Yes, but you must be careful with namespaces. The good point is that this method simplifies everything: you can load a pre-trained VGG-16, access any nodes in the graph, plug your own operations and train the whole thing!

If you only want to fine-tune your own nodes, you can stop the gradients anywhere you want, to avoid training the whole graph.

Graph mixing

**A closer real-life example**

I made a full example of a **Saver** usage in a more realistic setting here: <https://github.com/metaflow-ai/blog/blob/master/tf-save-load/embedding.py>

Don’t hesitate to run the code and see for yourself what happens.

**Annexe: More about the TF data ecosystem**

We are talking about Google here, and they mainly use in-house built tools when dealing with their work so it will be no surprise to discover that data are saved in the ProtoBuff format.

**Protocol buffers**

[Protocol Buffers often abbreviated Protobuf](https://developers.google.com/protocol-buffers/)s is the format used by TF to store and transfer data efficiently.

I don’t want to go into details but think about it as a faster JSON format that you can compress when you need to save space/bandwidth for storage/transfer. To recapitulate, you can use Protobufs as:

* An uncompressed, human-friendly, text format with the extension **.pbtxt**
* A compressed, machine friendly, binary format with the extension **.pb**or no extension at all

It’s like using JSON in your development setup and when moving to production, compressing your data on the fly for efficiency. Many more things can be done with Protobufs, if you are interested check the [tutorials here](https://developers.google.com/protocol-buffers/docs/pythontutorial)

*Neat trick: All operations dealing with Protobufs in TensorFlow have this****“\_def”****suffix that indicates “protocol buffer definition”. For example, to load the Protobufs of a saved graph, you can use the function:*tf.import\_graph\_def*. And to get the current graph as a Protobufs, you can use:*Graph.as\_graph\_def()*.*

**Files architecture**

Getting back to TF, when you save your data the usual way, you end up with 5 different type of files:

* A **“checkpoint”** file
* Some **“data”**files
* A **“meta”** file
* An **“index”** file
* If you use Tensorboard, an **“events”** file
* If you dump the human-friendly version: a**“textual Protobufs”** file

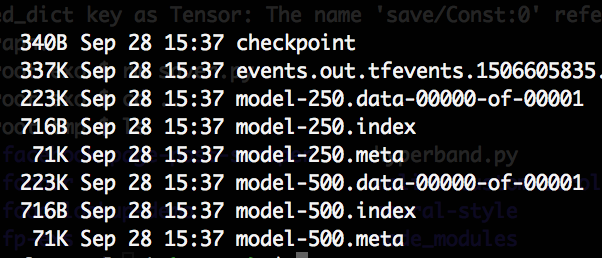
Let’s take a break here. When you think about it, what could potentially be saved when you are doing machine learning?

You can save the architecture of your model and the learned weights associated with it. You might want to save some training characteristics like the loss and accuracy of your model while training or even the whole training architecture. You might want to save hyperparameters and other operations to restart training later or replicate a result. **This is exactly what TensorFlow does.**

The three checkpoint files type are here to store the compressed data about your models and its weights.

* The **checkpoint** file is just a bookkeeping file that you can use in combination of high-level helper for loading different time saved chkp files.
* The **.meta** file holds the compressed Protobufs graph of your model and all the metadata associated (collections, learning rate, operations, etc.)
* The **.index**file holds an immutable key-value table linking a serialised tensor name and where to find its data in the **chkp.data files**
* The **.data**fileshold the data (weights) itself (this one is usually quite big in size). There can be many data files because they can be sharded and/or created on multiple timesteps while training.
* Finally, the **events** file store everything you need to visualise your model and all the data measured while you were training using summaries. This has nothing to do with saving/restoring your models itself.

Let’s have a look at the following screen capture of a result folder:



Screen capture of the resulting folder of some random training

* The weights filename is as follow: <prefix>-<global\_step>.data-<shard\_index>-of-<number\_of\_shards>.
* The model has been saved 2times at steps 250 and 500 in only 1 file (no shards).
* **data** files are a lot heavier than the **meta**files which is to be expected as they are containing the weights of our model
* The **index**file is very light as expected since it’s just a key-value table.

TF comes with multiple handy helpers like:

* Handling different checkpoints of your model in time and iteration. This can be a lifesaver if one of your machines break before the end of a training.
* Separating weights and metadata. You can share a model without its training weight easily.
* Saving metadata allows you to be sure to reproduce a result or continue a training etc.