Save and Restore

The `tf.train.Saver` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) class provides methods to save and restore models. The `tf.saved_model.simple_save` (https://www.tensorflow.org/api_docs/python/tf/saved_model/simple_save) function is an easy way to build a `tf.saved_model` (https://www.tensorflow.org/api_docs/python/tf/saved_model) suitable for serving. Estimators (https://www.tensorflow.org/guide/estimators) automatically save and restore variables in the `model_dir`.

Save and restore variables

TensorFlow Variables (https://www.tensorflow.org/guide/variables) are the best way to represent shared, persistent state manipulated by your program. The `tf.train.Saver` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) constructor adds `save` and `restore` ops to the graph for all, or a specified list, of the variables in the graph. The `Saver` object provides methods to run these ops, specifying paths for the checkpoint files to write to or read from.

`Saver` restores all variables already defined in your model. If you're loading a model without knowing how to build its graph (for example, if you're writing a generic program to load models), then read the Overview of saving and restoring models (#models) section later in this document.

TensorFlow saves variables in binary *checkpoint files* that map variable names to tensor values.

Caution: TensorFlow model files are code. Be careful with untrusted code. SeeUsing TensorFlow Securely (https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md)for details.

Save variables

Create a `Saver` with `tf.train.Saver()` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) to manage all variables in the model. For example, the following snippet demonstrates how to call the `tf.train.Saver.save` (https://www.tensorflow.org/api_docs/python/tf/train/Saver#save) method to save variables to checkpoint files:

```
# Create some variables.
v1 = tf.get_variable("v1", shape=[3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", shape=[5], initializer = tf.zeros_initializer)

inc_v1 = v1.assign(v1+1)
dec_v2 = v2.assign(v2-1)

# Add an op to initialize the variables.
init_op = tf.global_variables_initializer()

# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, initialize the variables, do some work, and save the
# variables to disk.
with tf.Session() as sess:
  sess.run(init_op)
  # Do some work with the model.
  inc_v1.op.run()
  dec_v2.op.run()
  # Save the variables to disk.
  save_path = saver.save(sess, "/tmp/model.ckpt")
  print("Model saved in path: %s" % save_path)
```

Restore variables

The `tf.train.Saver` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) object not only saves variables to checkpoint files, it also restores variables. Note that when you restore variables you do not have to initialize them beforehand. For example, the following snippet demonstrates how to call the `tf.train.Saver.restore`

(https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) method to restore variables from the checkpoint files:

```
tf.reset_default_graph()

# Create some variables.
v1 = tf.get_variable("v1", shape=[3])
v2 = tf.get_variable("v2", shape=[5])

# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, use the saver to restore variables from disk, and
# do some work with the model.
with tf.Session() as sess:
  # Restore variables from disk.
  saver.restore(sess, "/tmp/model.ckpt")
  print("Model restored.")
  # Check the values of the variables
  print("v1 : %s" % v1.eval())
  print("v2 : %s" % v2.eval())
```

Note: There is not a physical file called **`/tmp/model.ckpt`**. It is the *prefix* of filenames created for the checkpoint. Users only interact with the prefix instead of physical checkpoint files.

Choose variables to save and restore

If you do not pass any arguments to **`tf.train.Saver()`** (https://www.tensorflow.org/api_docs/python/tf/train/Saver), the saver handles all variables in the graph. Each variable is saved under the name that was passed when the variable was created.

It is sometimes useful to explicitly specify names for variables in the checkpoint files. For example, you may have trained a model with a variable named "**`weights`**" whose value you want to restore into a variable named "**`params`**".

It is also sometimes useful to only save or restore a subset of the variables used by a model. For example, you may have trained a neural net with five layers, and you

now want to train a new model with six layers that reuses the existing weights of the five trained layers. You can use the saver to restore the weights of just the first five layers.

You can easily specify the names and variables to save or load by passing to the `tf.train.Saver()` (https://www.tensorflow.org/api_docs/python/tf/train/Saver) constructor either of the following:

- A list of variables (which will be stored under their own names).
- A Python dictionary in which keys are the names to use and the values are the variables to manage.

Continuing from the save/restore examples shown earlier:

```
tf.reset_default_graph()
# Create some variables.
v1 = tf.get_variable("v1", [3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", [5], initializer = tf.zeros_initializer)

# Add ops to save and restore only `v2` using the name "v2"
saver = tf.train.Saver({"v2": v2})

# Use the saver object normally after that.
with tf.Session() as sess:
  # Initialize v1 since the saver will not.
  v1.initializer.run()
  saver.restore(sess, "/tmp/model.ckpt")

  print("v1 : %s" % v1.eval())
  print("v2 : %s" % v2.eval())
```

Notes:

- You can create as many `Saver` objects as you want if you need to save and restore different subsets of the model variables. The same variable can be listed in multiple saver objects; its value is only changed when the `Saver.restore()` (https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) method is run.

- If you only restore a subset of the model variables at the start of a session, you have to run an initialize op for the other variables. See `tf.variables_initializer` (https://www.tensorflow.org/api_docs/python/tf/initializers/variables) for more information.
- To inspect the variables in a checkpoint, you can use the `inspect_checkpoint` (https://www.tensorflow.org/code/tensorflow/python/tools/inspect_checkpoint.py) library, particularly the `print_tensors_in_checkpoint_file` function.
- By default, `Saver` uses the value of the `tf.Variable.name` (https://www.tensorflow.org/api_docs/python/tf/Variable#name) property for each variable. However, when you create a `Saver` object, you may optionally choose names for the variables in the checkpoint files.

Inspect variables in a checkpoint

We can quickly inspect variables in a checkpoint with the `inspect_checkpoint` (https://www.tensorflow.org/code/tensorflow/python/tools/inspect_checkpoint.py) library.

Continuing from the save/restore examples shown earlier:

```
# import the inspect_checkpoint library
from tensorflow.python.tools import inspect_checkpoint as chkp

# print all tensors in checkpoint file
chkp.print_tensors_in_checkpoint_file("/tmp/model.ckpt", tensor_name='', all_ten

# tensor_name:  v1
# [ 1.  1.  1.]
# tensor_name:  v2
# [-1. -1. -1. -1. -1.]

# print only tensor v1 in checkpoint file
chkp.print_tensors_in_checkpoint_file("/tmp/model.ckpt", tensor_name='v1', all_t

# tensor_name:  v1
# [ 1.  1.  1.]

# print only tensor v2 in checkpoint file
```

```
chkp.print_tensors_in_checkpoint_file("/tmp/model.ckpt", tensor_name='v2', all_t

# tensor_name:  v2
# [-1. -1. -1. -1. -1.]
```

Save and restore models

Use **`SavedModel`** to save and load your model—variables, the graph, and the graph's metadata. This is a language-neutral, recoverable, hermetic serialization format that enables higher-level systems and tools to produce, consume, and transform TensorFlow models. TensorFlow provides several ways to interact with **`SavedModel`**, including the **`tf.saved_model`** (https://www.tensorflow.org/api_docs/python/tf/saved_model) APIs, **`tf.estimator.Estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator), and a command-line interface.

Build and load a SavedModel

Simple save

The easiest way to create a **`SavedModel`** is to use the **`tf.saved_model.simple_save`** (https://www.tensorflow.org/api_docs/python/tf/saved_model/simple_save) function:

```
simple_save(session,
            export_dir,
            inputs={"x": x, "y": y},
            outputs={"z": z})
```

This configures the **`SavedModel`** so it can be loaded by TensorFlow serving (https://www.tensorflow.org/tfx/tutorials/serving/rest_simple) and supports the Predict API (https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis/predict.proto).

To access the classify, regress, or multi-inference APIs, use the manual `SavedModel` builder APIs or an `tf.estimator.Estimator` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator).

Manually build a SavedModel

If your use case isn't covered by `tf.saved_model.simple_save` (https://www.tensorflow.org/api_docs/python/tf/saved_model/simple_save), use the manual `tf.saved_model.builder` (https://www.tensorflow.org/api_docs/python/tf/saved_model/builder) to create a `SavedModel`.

The `tf.saved_model.builder.SavedModelBuilder` (https://www.tensorflow.org/api_docs/python/tf/saved_model/Builder) class provides functionality to save multiple `MetaGraphDef`s. A **MetaGraph** is a dataflow graph, plus its associated variables, assets, and signatures. A `MetaGraphDef` is the protocol buffer representation of a MetaGraph. A **signature** is the set of inputs to and outputs from a graph.

If assets need to be saved and written or copied to disk, they can be provided when the first `MetaGraphDef` is added. If multiple `MetaGraphDef`s are associated with an asset of the same name, only the first version is retained.

Each `MetaGraphDef` added to the SavedModel must be annotated with user-specified tags. The tags provide a means to identify the specific `MetaGraphDef` to load and restore, along with the shared set of variables and assets. These tags typically annotate a `MetaGraphDef` with its functionality (for example, serving or training), and optionally with hardware-specific aspects (for example, GPU).

For example, the following code suggests a typical way to use `SavedModelBuilder` to build a SavedModel:

```
export_dir = ...
...
builder = tf.saved_model.builder.SavedModelBuilder(export_dir)
with tf.Session(graph=tf.Graph()) as sess:
  ...
  builder.add_meta_graph_and_variables(sess,
                                       [tag_constants.TRAINING],
```

```
                                        signature_def_map=foo_signatures,
                                        assets_collection=foo_assets,
                                        strip_default_attrs=True)
...
# Add a second MetaGraphDef for inference.
with tf.Session(graph=tf.Graph()) as sess:
  ...
  builder.add_meta_graph([tag_constants.SERVING], strip_default_attrs=True)
...
builder.save()
```

Forward compatibility via `strip_default_attrs=True`

Following the guidance below gives you forward compatibility only if the set of Ops has not changed.

The **`tf.saved_model.builder.SavedModelBuilder`** (https://www.tensorflow.org/api_docs/python/tf/saved_model/Builder) class allows users to control whether default-valued attributes must be stripped from the **`NodeDefs`** (https://www.tensorflow.org/extend/tool_developers/index#nodes) while adding a meta graph to the SavedModel bundle. Both **`tf.saved_model.builder.SavedModelBuilder.add_meta_graph_and_variables`** (https://www.tensorflow.org/api_docs/python/tf/saved_model/Builder#add_meta_graph_and_variables) and **`tf.saved_model.builder.SavedModelBuilder.add_meta_graph`** (https://www.tensorflow.org/api_docs/python/tf/saved_model/Builder#add_meta_graph) methods accept a Boolean flag **`strip_default_attrs`** that controls this behavior.

If **`strip_default_attrs`** is **`False`**, the exported **`tf.MetaGraphDef`** (https://www.tensorflow.org/api_docs/python/tf/MetaGraphDef) will have the default valued attributes in all its **`tf.NodeDef`** (https://www.tensorflow.org/api_docs/python/tf/NodeDef) instances. This can break forward compatibility with a sequence of events such as the following:

- An existing Op (**`Foo`**) is updated to include a new attribute (**`T`**) with a default (**`bool`**) at version 101.
- A model producer such as a "trainer binary" picks up this change (version 101) to the **`OpDef`** and re-exports an existing model that uses Op **`Foo`**.

- A model consumer (such as Tensorflow Serving (/serving)) running an older binary (version 100) doesn't have attribute `T` for Op `Foo`, but tries to import this model. The model consumer doesn't recognize attribute `T` in a `NodeDef` that uses Op `Foo` and therefore fails to load the model.
- By setting `strip_default_attrs` to True, the model producers can strip away any default valued attributes in the `NodeDefs`. This helps ensure that newly added attributes with defaults don't cause older model consumers to fail loading models regenerated with newer training binaries.

See compatibility guidance (https://www.tensorflow.org/guide/version_compat) for more information.

Loading a SavedModel in Python

The Python version of the SavedModel `tf.saved_model.loader` (https://www.tensorflow.org/api_docs/python/tf/saved_model/loader) provides load and restore capability for a SavedModel. The `load` operation requires the following information:

- The session in which to restore the graph definition and variables.
- The tags used to identify the MetaGraphDef to load.
- The location (directory) of the SavedModel.

Upon a load, the subset of variables, assets, and signatures supplied as part of the specific MetaGraphDef will be restored into the supplied session.

```
export_dir = ...
...
with tf.Session(graph=tf.Graph()) as sess:
  tf.saved_model.loader.load(sess, [tag_constants.TRAINING], export_dir)
  ...
```

Load a SavedModel in C++

The C++ version of the SavedModel loader (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved_model/loader.h) provides an API to load a SavedModel from a path, while allowing **`SessionOptions`** and **`RunOptions`**. You have to specify the tags associated with the graph to be loaded. The loaded version of SavedModel is referred to as **`SavedModelBundle`** and contains the MetaGraphDef and the session within which it is loaded.

```
const string export_dir = ...
SavedModelBundle bundle;
...
LoadSavedModel(session_options, run_options, export_dir, {kSavedModelTagTrain},
               &bundle);
```

Load and serve a SavedModel in TensorFlow serving

You can easily load and serve a SavedModel with the TensorFlow Serving Model Server binary. See instructions (https://www.tensorflow.org/tfx/serving/setup) on how to install the server, or build it if you wish.

Once you have the Model Server, run it with:

```
tensorflow_model_server --port=port-numbers --model_name=your-model-name --model
```

Set the port and model_name flags to values of your choosing. The model_base_path flag expects to be to a base directory, with each version of your model residing in a numerically named subdirectory. If you only have a single version of your model, simply place it in a subdirectory like so:

- Place the model in /tmp/model/0001
- Set model_base_path to /tmp/model

Store different versions of your model in numerically named subdirectories of a common base directory. For example, suppose the base directory is **`/tmp/model`**. If you have only one version of your model, store it in **`/tmp/model/0001`**. If you have two versions of your model, store the second version in **`/tmp/model/0002`**, and so

on. Set the `--model-base_path` flag to the base directory (`/tmp/model`, in this example). TensorFlow Model Server will serve the model in the highest numbered subdirectory of that base directory.

Standard constants

SavedModel offers the flexibility to build and load TensorFlow graphs for a variety of use-cases. For the most common use-cases, SavedModel's APIs provide a set of constants in Python and C++ that are easy to reuse and share across tools consistently.

Standard MetaGraphDef tags

You may use sets of tags to uniquely identify a `MetaGraphDef` saved in a SavedModel. A subset of commonly used tags is specified in:

- Python (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/tag_constants.py)
- C++ (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved_model/tag_constants.h)

Standard SignatureDef constants

A **SignatureDef** (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/meta_graph.proto) is a protocol buffer that defines the signature of a computation supported by a graph. Commonly used input keys, output keys, and method names are defined in:

- Python (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/signature_constants.py)
- C++ (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/saved_model/signature_constants.h)

Using SavedModel with Estimators

After training an `Estimator` model, you may want to create a service from that model that takes requests and returns a result. You can run such a service locally on your machine or deploy it in the cloud.

To prepare a trained Estimator for serving, you must export it in the standard SavedModel format. This section explains how to:

- Specify the output nodes and the corresponding APIs (https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis/prediction_service.proto) that can be served (Classify, Regress, or Predict).
- Export your model to the SavedModel format.
- Serve the model from a local server and request predictions.

Prepare serving inputs

During training, an `input_fn()` (https://www.tensorflow.org/guide/premade_estimators#input_fn) ingests data and prepares it for use by the model. At serving time, similarly, a `serving_input_receiver_fn()` accepts inference requests and prepares them for the model. This function has the following purposes:

- To add placeholders to the graph that the serving system will feed with inference requests.
- To add any additional ops needed to convert data from the input format into the feature `Tensor`s expected by the model.

The function returns a `tf.estimator.export.ServingInputReceiver` (https://www.tensorflow.org/api_docs/python/tf/estimator/export/ServingInputReceiver) object, which packages the placeholders and the resulting feature `Tensor`s together.

A typical pattern is that inference requests arrive in the form of serialized `tf.Example`s, so the `serving_input_receiver_fn()` creates a single string placeholder to receive them. The `serving_input_receiver_fn()` is then also

responsible for parsing the `tf.Example`s by adding a `tf.parse_example` (https://www.tensorflow.org/api_docs/python/tf/io/parse_example) op to the graph.

When writing such a `serving_input_receiver_fn()`, you must pass a parsing specification to `tf.parse_example` (https://www.tensorflow.org/api_docs/python/tf/io/parse_example) to tell the parser what feature names to expect and how to map them to `Tensor`s. A parsing specification takes the form of a dict from feature names to `tf.FixedLenFeature` (https://www.tensorflow.org/api_docs/python/tf/io/FixedLenFeature), `tf.VarLenFeature` (https://www.tensorflow.org/api_docs/python/tf/io/VarLenFeature), and `tf.SparseFeature` (https://www.tensorflow.org/api_docs/python/tf/io/SparseFeature). Note this parsing specification should not include any label or weight columns, since those will not be available at serving time—in contrast to a parsing specification used in the `input_fn()` at training time.

In combination, then:

```
feature_spec = {'foo': tf.FixedLenFeature(...),
                'bar': tf.VarLenFeature(...)}

def serving_input_receiver_fn():
  """An input receiver that expects a serialized tf.Example."""
  serialized_tf_example = tf.placeholder(dtype=tf.string,
                                         shape=[default_batch_size],
                                         name='input_example_tensor')
  receiver_tensors = {'examples': serialized_tf_example}
  features = tf.parse_example(serialized_tf_example, feature_spec)
  return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)
```

The `tf.estimator.export.build_parsing_serving_input_receiver_fn` (https://www.tensorflow.org/api_docs/python/tf/estimator/export/build_parsing_serving_input_receiver_fn) utility function provides that input receiver for the common case.

★ **Note:** when training a model to be served using the Predict API with a local server, the parsing step is not needed because the model will receive raw feature data.

Even if you require no parsing or other input processing—that is, if the serving system will feed feature `Tensor`s directly—you must still provide a `serving_input_receiver_fn()` that creates placeholders for the feature `Tensor`s and passes them through. The `tf.estimator.export.build_raw_serving_input_receiver_fn` (https://www.tensorflow.org/api_docs/python/tf/estimator/export/build_raw_serving_input_receiver_fn) utility provides for this.

If these utilities do not meet your needs, you are free to write your own `serving_input_receiver_fn()`. One case where this may be needed is if your training `input_fn()` incorporates some preprocessing logic that must be recapitulated at serving time. To reduce the risk of training-serving skew, we recommend encapsulating such processing in a function which is then called from both `input_fn()` and `serving_input_receiver_fn()`.

Note that the `serving_input_receiver_fn()` also determines the *input* portion of the signature. That is, when writing a `serving_input_receiver_fn()`, you must tell the parser what signatures to expect and how to map them to your model's expected inputs. By contrast, the *output* portion of the signature is determined by the model.

Specify the outputs of a custom model

When writing a custom `model_fn`, you must populate the `export_outputs` element of the `tf.estimator.EstimatorSpec` (https://www.tensorflow.org/api_docs/python/tf/estimator/EstimatorSpec) return value. This is a dict of `{name: output}` describing the output signatures to be exported and used during serving.

In the usual case of making a single prediction, this dict contains one element, and the `name` is immaterial. In a multi-headed model, each head is represented by an entry in this dict. In this case the `name` is a string of your choice that can be used to request a specific head at serving time.

Each `output` value must be an `ExportOutput` object such as `tf.estimator.export.ClassificationOutput` (https://www.tensorflow.org/api_docs/python/tf/estimator/export/ClassificationOutput), `tf.estimator.export.RegressionOutput`

(https://www.tensorflow.org/api_docs/python/tf/estimator/export/RegressionOutput), or `tf.estimator.export.PredictOutput` (https://www.tensorflow.org/api_docs/python/tf/estimator/export/PredictOutput).

These output types map straightforwardly to the TensorFlow Serving APIs (https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis/prediction_service.proto), and so determine which request types will be honored.

Note: In the multi-headed case, a **`SignatureDef`** will be generated for each element of the **`export_outputs`** dict returned from the model_fn, named using the same keys. These **`SignatureDef`**s differ only in their outputs, as provided by the corresponding **`ExportOutput`** entry. The inputs are always those provided by the **`serving_input_receiver_fn`**. An inference request may specify the head by name. One head must be named using (https://www.tensorflow.org/code/tensorflow/python/saved_model/signature_constants.py) **`signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY`** (https://www.tensorflow.org/api_docs/python/tf/saved_model/signature_constants#DEFAULT_SERVING_SIGNATURE_DEF_KEY) indicating which **`SignatureDef`** will be served when an inference request does not specify one.

Perform the export

To export your trained Estimator, call `tf.estimator.Estimator.export_savedmodel` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator#export_savedmodel) with the export base path and the `serving_input_receiver_fn`.

```
estimator.export_savedmodel(export_dir_base, serving_input_receiver_fn,
                            strip_default_attrs=True)
```

This method builds a new graph by first calling the `serving_input_receiver_fn()` to obtain feature `Tensor`s, and then calling this `Estimator`'s `model_fn()` to generate the model graph based on those features. It starts a fresh `Session`, and, by default, restores the most recent checkpoint into it. (A different checkpoint may be passed, if needed.) Finally it creates a time-stamped export directory below the given **`export_dir_base`** (i.e.,

`export_dir_base/<timestamp>`), and writes a SavedModel into it containing a single `MetaGraphDef` saved from this Session.

★ **Note:** It is your responsibility to garbage-collect old exports. Otherwise, successive exports will accumulate under `export_dir_base`.

Serve the exported model locally

For local deployment, you can serve your model using TensorFlow Serving (https://github.com/tensorflow/serving), an open-source project that loads a SavedModel and exposes it as a gRPC (https://www.grpc.io/) service.

First, install TensorFlow Serving (https://github.com/tensorflow/serving).

Then build and run the local model server, substituting `$export_dir_base` with the path to the SavedModel you exported above:

```
bazel build //tensorflow_serving/model_servers:tensorflow_model_server
bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server --port=9000 -
```

Now you have a server listening for inference requests via gRPC on port 9000!

Request predictions from a local server

The server responds to gRPC requests according to the PredictionService (https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis/prediction_service.proto#L15)
gRPC API service definition. (The nested protocol buffers are defined in various neighboring files (https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis)).

From the API service definition, the gRPC framework generates client libraries in various languages providing remote access to the API. In a project using the Bazel build tool, these libraries are built automatically and provided via dependencies like these (using Python for example):

```
  deps = [
    "//tensorflow_serving/apis:classification_proto_py_pb2",
    "//tensorflow_serving/apis:regression_proto_py_pb2",
    "//tensorflow_serving/apis:predict_proto_py_pb2",
    "//tensorflow_serving/apis:prediction_service_proto_py_pb2"
  ]
```

Python client code can then import the libraries thus:

```
from tensorflow_serving.apis import classification_pb2
from tensorflow_serving.apis import regression_pb2
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2
```

★ **Note:** **`prediction_service_pb2`** defines the service as a whole and so is always required. However a typical client will need only one of **`classification_pb2`**, **`regression_pb2`**, and **`predict_pb2`**, depending on the type of requests being made.

Sending a gRPC request is then accomplished by assembling a protocol buffer containing the request data and passing it to the service stub. Note how the request protocol buffer is created empty and then populated via the generated protocol buffer API (https://developers.google.com/protocol-buffers/docs/reference/python-generated).

```
from grpc.beta import implementations

channel = implementations.insecure_channel(host, int(port))
stub = prediction_service_pb2.beta_create_PredictionService_stub(channel)

request = classification_pb2.ClassificationRequest()
example = request.input.example_list.examples.add()
example.features.feature['x'].float_list.value.extend(image[0].astype(float))

result = stub.Classify(request, 10.0)  # 10 secs timeout
```

The returned result in this example is a **`ClassificationResponse`** protocol buffer.

This is a skeletal example; please see the Tensorflow Serving (https://www.tensorflow.org/deploy/index) documentation and examples (https://github.com/tensorflow/serving/tree/master/tensorflow_serving/example) for more details.

★ **Note:** **`ClassificationRequest`** and **`RegressionRequest`** contain a **`tensorflow.serving.Input`** protocol buffer, which in turn contains a list of **`tensorflow.Example`** protocol buffers. **`PredictRequest`**, by contrast, contains a mapping from feature names to values encoded via **`TensorProto`**. Correspondingly: When using the **`Classify`** and **`Regress`** APIs, TensorFlow Serving feeds serialized **`tf.Example`**s to the graph, so your **`serving_input_receiver_fn()`** should include a **`tf.parse_example()`** (https://www.tensorflow.org/api_docs/python/tf/io/parse_example) Op. When using the generic **`Predict`** API, however, TensorFlow Serving feeds raw feature data to the graph, so a pass through **`serving_input_receiver_fn()`** should be used.

CLI to inspect and execute SavedModel

You can use the SavedModel Command Line Interface (CLI) to inspect and execute a SavedModel. For example, you can use the CLI to inspect the model's **`SignatureDef`**s. The CLI enables you to quickly confirm that the input Tensor dtype and shape (https://www.tensorflow.org/guide/tensors) match the model. Moreover, if you want to test your model, you can use the CLI to do a sanity check by passing in sample inputs in various formats (for example, Python expressions) and then fetching the output.

Install the SavedModel CLI

Broadly speaking, you can install TensorFlow in either of the following two ways:

- By installing a pre-built TensorFlow binary.
- By building TensorFlow from source code.

If you installed TensorFlow through a pre-built TensorFlow binary, then the SavedModel CLI is already installed on your system at pathname **`bin\saved_model_cli`**.

If you built TensorFlow from source code, you must run the following additional command to build **`saved_model_cli`**:

```
$ bazel build tensorflow/python/tools:saved_model_cli
```

Overview of commands

The SavedModel CLI supports the following two commands on a **`MetaGraphDef`** in a SavedModel:

- **`show`**, which shows a computation on a **`MetaGraphDef`** in a SavedModel.
- **`run`**, which runs a computation on a **`MetaGraphDef`**.

`show` command

A SavedModel contains one or more **`MetaGraphDef`**s, identified by their tag-sets. To serve a model, you might wonder what kind of **`SignatureDef`**s are in each model, and what are their inputs and outputs. The **`show`** command let you examine the contents of the SavedModel in hierarchical order. Here's the syntax:

```
usage: saved_model_cli show [-h] --dir DIR [--all]
[--tag_set TAG_SET] [--signature_def SIGNATURE_DEF_KEY]
```

For example, the following command shows all available MetaGraphDef tag-sets in the SavedModel:

```
$ saved_model_cli show --dir /tmp/saved_model_dir
The given SavedModel contains the following tag-sets:
serve
serve, gpu
```

The following command shows all available **`SignatureDef`** keys in a **`MetaGraphDef`**:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --tag_set serve
The given SavedModel `MetaGraphDef` contains `SignatureDefs` with the
following keys:
SignatureDef key: "classify_x2_to_y3"
SignatureDef key: "classify_x_to_y"
SignatureDef key: "regress_x2_to_y3"
SignatureDef key: "regress_x_to_y"
SignatureDef key: "regress_x_to_y2"
SignatureDef key: "serving_default"
```

If a **`MetaGraphDef`** has *multiple* tags in the tag-set, you must specify all tags, each tag separated by a comma. For example:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --tag_set serve,gpu
```

To show all inputs and outputs TensorInfo for a specific **`SignatureDef`**, pass in the **`SignatureDef`** key to **`signature_def`** option. This is very useful when you want to know the tensor key value, dtype and shape of the input tensors for executing the computation graph later. For example:

```
$ saved_model_cli show --dir \
/tmp/saved_model_dir --tag_set serve --signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['x'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: x:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['y'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: y:0
Method name is: tensorflow/serving/predict
```

To show all available information in the SavedModel, use the `--all` option. For example:

```
$ saved_model_cli show --dir /tmp/saved_model_dir --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['classify_x2_to_y3']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['inputs'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: x2:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['scores'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: y3:0
  Method name is: tensorflow/serving/classify

...

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['x'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: x:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['y'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: y:0
  Method name is: tensorflow/serving/predict
```

`run` command

Invoke the `run` command to run a graph computation, passing inputs and then displaying (and optionally saving) the outputs. Here's the syntax:

```
usage: saved_model_cli run [-h] --dir DIR --tag_set TAG_SET --signature_def
                           SIGNATURE_DEF_KEY [--inputs INPUTS]
                           [--input_exprs INPUT_EXPRS]
                           [--input_examples INPUT_EXAMPLES] [--outdir OUTDIR]
                           [--overwrite] [--tf_debug]
```

The `run` command provides the following three ways to pass inputs to the model:

- `--inputs` option enables you to pass numpy ndarray in files.
- `--input_exprs` option enables you to pass Python expressions.
- `--input_examples` option enables you to pass `tf.train.Example`.

`--inputs`

To pass input data in files, specify the `--inputs` option, which takes the following general format:

```
--inputs <INPUTS>
```

where *INPUTS* is either of the following formats:

- `<input_key>=<filename>`
- `<input_key>=<filename>[<variable_name>]`

You may pass multiple *INPUTS*. If you do pass multiple inputs, use a semicolon to separate each of the *INPUTS*.

`saved_model_cli` uses `numpy.load` to load the *filename*. The *filename* may be in any of the following formats:

- `.npy`
- `.npz`
- pickle format

A **.npy** file always contains a numpy ndarray. Therefore, when loading from a **.npy** file, the content will be directly assigned to the specified input tensor. If you specify a *variable_name* with that **.npy** file, the *variable_name* will be ignored and a warning will be issued.

When loading from a **.npz** (zip) file, you may optionally specify a *variable_name* to identify the variable within the zip file to load for the input tensor key. If you don't specify a *variable_name*, the SavedModel CLI will check that only one file is included in the zip file and load it for the specified input tensor key.

When loading from a pickle file, if no **variable_name** is specified in the square brackets, whatever that is inside the pickle file will be passed to the specified input tensor key. Otherwise, the SavedModel CLI will assume a dictionary is stored in the pickle file and the value corresponding to the *variable_name* will be used.

--input_exprs

To pass inputs through Python expressions, specify the **--input_exprs** option. This can be useful for when you don't have data files lying around, but still want to sanity check the model with some simple inputs that match the dtype and shape of the model's **SignatureDef**s. For example:

```
`<input_key>=[[1],[2],[3]]`
```

In addition to Python expressions, you may also pass numpy functions. For example:

```
`<input_key>=np.ones((32,32,3))`
```

(Note that the **numpy** module is already available to you as **np**.)

--input_examples

To pass **tf.train.Example** (https://www.tensorflow.org/api_docs/python/tf/train/Example) as inputs, specify the **--input_examples** option. For each input key, it takes a list of dictionary, where each

dictionary is an instance of **`tf.train.Example`** (https://www.tensorflow.org/api_docs/python/tf/train/Example). The dictionary keys are the features and the values are the value lists for each feature. For example:

```
`<input_key>=[{"age":[22,24],"education":["BS","MS"]}]`
```

Save output

By default, the SavedModel CLI writes output to stdout. If a directory is passed to `--outdir` option, the outputs will be saved as npy files named after output tensor keys under the given directory.

Use `--overwrite` to overwrite existing output files.

TensorFlow debugger (tfdbg) integration

If `--tf_debug` option is set, the SavedModel CLI will use the TensorFlow Debugger (tfdbg) to watch the intermediate Tensors and runtime graphs or subgraphs while running the SavedModel.

Full examples of `run`

Given:

- Your model simply adds `x1` and `x2` to get output `y`.
- All tensors in the model have shape `(-1, 1)`.
- You have two `npy` files:
  - `/tmp/my_data1.npy`, which contains a numpy ndarray `[[1], [2], [3]]`.
  - `/tmp/my_data2.npy`, which contains another numpy ndarray `[[0.5], [0.5], [0.5]]`.

To run these two `npy` files through the model to get output `y`, issue the following command:

```
$ saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
--signature_def x1_x2_to_y --inputs 'x1=/tmp/my_data1.npy;x2=/tmp/my_data2.npy'
--outdir /tmp/out
Result for output key y:
[[ 1.5]
 [ 2.5]
 [ 3.5]]
```

Let's change the preceding example slightly. This time, instead of two `.npy` files, you now have an `.npz` file and a pickle file. Furthermore, you want to overwrite any existing output file. Here's the command:

```
$ saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
--signature_def x1_x2_to_y \
--inputs 'x1=/tmp/my_data1.npz[x];x2=/tmp/my_data2.pkl' --outdir /tmp/out \
--overwrite
Result for output key y:
[[ 1.5]
 [ 2.5]
 [ 3.5]]
```

You may specify python expression instead of an input file. For example, the following command replaces input `x2` with a Python expression:

```
$ saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
--signature_def x1_x2_to_y --inputs x1=/tmp/my_data1.npz[x] \
--input_exprs 'x2=np.ones((3,1))'
Result for output key y:
[[ 2]
 [ 3]
 [ 4]]
```

To run the model with the TensorFlow Debugger on, issue the following command:

```
$ saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
--signature_def serving_default --inputs x=/tmp/data.npz[x] --tf_debug
```

Structure of a SavedModel directory

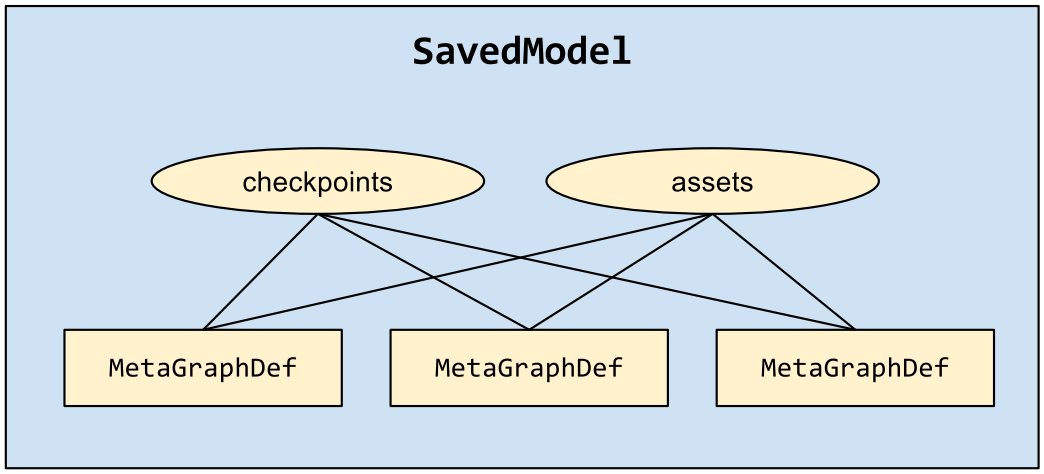
When you save a model in SavedModel format, TensorFlow creates a SavedModel directory consisting of the following subdirectories and files:

```
assets/
assets.extra/
variables/
    variables.data-?????-of-?????
    variables.index
saved_model.pb|saved_model.pbtxt
```

where:

- `assets` is a subfolder containing auxiliary (external) files, such as vocabularies. Assets are copied to the SavedModel location and can be read when loading a specific `MetaGraphDef`.
- `assets.extra` is a subfolder where higher-level libraries and users can add their own assets that co-exist with the model, but are not loaded by the graph. This subfolder is not managed by the SavedModel libraries.
- `variables` is a subfolder that includes output from `tf.train.Saver` (https://www.tensorflow.org/api_docs/python/tf/train/Saver).
- `saved_model.pb` or `saved_model.pbtxt` is the SavedModel protocol buffer. It includes the graph definitions as `MetaGraphDef` protocol buffers.

A single SavedModel can represent multiple graphs. In this case, all the graphs in the SavedModel share a *single* set of checkpoints (variables) and assets. For example, the following diagram shows one SavedModel containing three `MetaGraphDef`s, all three of which share the same set of checkpoints and assets:



Each graph is associated with a specific set of tags, which enables identification during a load or restore operation.