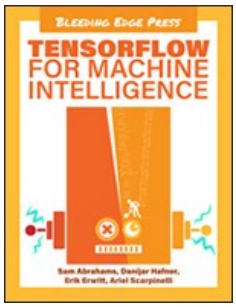

TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli

---

Chapter 7: Deploying Models in Production

Overview

So far we have seen how to work with Tensorflow for building and training models from basic machine learning to complex deep learning networks. In this chapter we are going to focus on putting our trained models in production so they can be consumed by other apps.

Our goal will be to create a simple webapp that will allow the user to upload an image and run the Inception model over it for classifying.

Setting up a Tensorflow serving development environment

Tensorflow serving is the tool for building servers that allow to use our models in production. There are two flavors to use it during development: manually installing all the dependencies and tools for building it from source, or using a Docker image. We are going to use the latter since it is easier, cleaner, and allows you to develop in other environments than Linux.

In case you don't know what a Docker image is, think of it as a lightweight version of a virtual machine image that runs without the overhead of running a full OS inside it. You should install Docker in your development machine first if you haven't. Follow instructions from https://docs.docker.com/engine/installation/.

To use the Docker image, we have available the https://github.com/tensorflow/serving/blob/master/tensorflow_serving/tools/docker/Dockerfile.devel file, which is the configuration file to create the image locally, so to use it we should:

```
docker build --pull -t $USER/tensorflow-serving-devel https://raw.githubuser-
content.com/tensorflow/serving/master/tensorflow_serving/tools/docker/Docker-
file.devel
```

Be aware that it may take a while to download all of the dependencies.

Now to run the container using the image to start working on it we use:

```
docker run -v $HOME:/mnt/home -p 9999:9999 -it $USER/tensorflow-serving-devel
```

That will load mount your home directory in the `/mnt/home` path of the container, and will let you working in a terminal inside of it. This is useful as you can work your code directly on your favorite IDE/editor, and just use the container for running the build tools. It will also leave the port 9999 open to access it from your host machine for later usage of the server we are going to build.

You can leave the container terminal with `exit`, which will stop it from running, and start it again as many times you want using command above.

Bazel workspace

Tensorflow Serving programs are coded in C++ and should be built using Google's Bazel build tool. We are going to run Bazel from inside the recently created container.

Bazel manages third party dependencies at code level, downloading and building them, as long as they are also built with Bazel. To define which third party dependencies our project would support, we must define a `WORKSPACE` file at the root of our project repository.

The dependencies we need are Tensorflow Serving repository, and for the case of our example, the Tensorflow Models repository includes the Inception model code.

Sadly, at the moment of this writing, Tensorflow Serving does not support being referenced directly thru Bazel as a Git repository, so we must include it as a Git submodule in our project:

```
# on your local machine
mkdir ~/serving_example
cd ~/serving_example
git init
git submodule add https://github.com/tensorflow/serving.git tf_serving
git submodule update --init --recursive
```

We now define the third party dependencies as locally stored files using the `local_repository` rule on the `WORKSPACE` file. We also have to initialize Tensorflow dependencies using the `tf_workspace` rule imported from the project:

```
# Bazel WORKSPACE file

workspace(name = "serving")

local_repository(
    name = "tf_serving",
    path = __workspace_dir__ + "/tf_serving",
)
```

```
local_repository(
    name = "org_tensorflow",
    path = __workspace_dir__ + "/tf_serving/tensorflow",
)

load('//tf_serving/tensorflow/tensorflow:workspace.bzl', 'tf_workspace')
tf_workspace("tf_serving/tensorflow/", "@org_tensorflow")

bind(
    name = "libssl",
    actual = "@boringssl_git//:ssl",
)

bind(
    name = "zlib",
    actual = "@zlib_archive//:zlib",
)

# only needed for inception model export
local_repository(
    name = "inception_model",
    path = __workspace_dir__ + "/tf_serving/tf_models/inception",
)
```

As a last step we have to run `./configure` for Tensorflow from within the container:

```
# on the docker container
cd /mnt/home/serving_example/tf_serving/tensorflow
./configure
```

Exporting trained models

Once our model is trained and we are happy with the evaluation, we will need to export its computing graph along its variables values in order to make it available for production usage.

The graph of the model should be slighlty changed from its training version, as it must take its inputs from placeholders and run a single step of inference on them to compute the output. For the example of the Inception model, and for any image recognition model in general, we want the input to be a single string representing a JPEG encoded image, so we can easily send it from our consumer app. This is quite different from the training input that reads from a TFRecords file.

The general form for defining the inputs should look like:

```
def convert_external_inputs(external_x):
   # transform the external input to the input format required on inference

def inference(x):
   # from the original model...

external_x = tf.placeholder(tf.string)
x = convert_external_inputs(external_x)
y = inference(x)
```

In the code above we define the placeholder for the input. We call a function to convert the external input represented in the placeholder to the format required for the original model inference method. For example we will convert from the JPEG string to the image format required for Inception. Finally we call the original model inference method with the converted input.

For example, for the Inception model we should have methods like:

```
import tensorflow as tf
from tensorflow_serving.session_bundle import exporter
from inception import inception_model

def convert_external_inputs(external_x):
    # transform the external input to the input format required on inference
    # convert the image string to a pixels tensor with values in the range
0,1
    image| = tf.image.convert_image_dtype(tf.image.decode_jpeg(external_x,
channels=3), tf.float32)
    # resize the image to the model expected width and height
    images = tf.image.resize_bilinear(tf.expand_dims(image, 0), [299, 299])
    # Convert the pixels to the range -1,1 required by the model
```

```
    images = tf.mul(tf.sub(images, 0.5), 2)
    return images

def inference(images):
   logits, _ = inception_model.inference(images, 1001)
   return logits
```

In the code above we define the placeholder for the input. We call a function to convert the external input represented in the placeholder to the format required for the original model inference method. For example we will convert from the JPEG string to the image format required for Inception. Finally we call the original model inference method with the converted input.

The inference method requires values for its parameters. We will recover those from a training checkpoint. As you may recall from the basics chapter, we periodically save training checkpoint files of our model. Those contain the learned values of parameters at the time, so in case of dissaster we don't loose the training progress.

When we declare the training complete, the last saved training checkpoint will contain the most updated model parameters, which are the ones we wish to put in production.

To restore the checkpoint, the code should be:

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

with tf.Session() as sess:
    # Restore variables from training checkpoints.
    ckpt = tf.train.get_checkpoint_state(sys.argv[1])
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, sys.argv[1] + "/" + ckpt.model_checkpoint_path)
    else:
        print("Checkpoint file not found")
        raise SystemExit
```

For the Inception model, you can download a pretrained checkpoint from http://download.tensorflow.org/models/image/imagenet/ inception-v3-2016-03-01.tar.gz

```
# on the docker container
cd /tmp
curl -O http://download.tensorflow.org/models/image/imagenet/inception-
v3-2016-03-01.tar.gz
tar -xzf inception-v3-2016-03-01.tar.gz
```

Finally we export the model using the `tensorflow_serving.session_bundle.exporter.Exporter` class. We create an instance of it passing the saver instance. Then we have to create the signature of the model using the `exporter.classification_signature` method. The signature specifices which is the `input_tensor`, and which are the output tensors. The output is composed by the `classes_tensor`, which contains the list of output class names, and the `scores_tensor`, which contains the score/probability the model assigns to each class. Tipically in a model with a high number of classes, you would configure those to return only classes selected with `tf.nn.top_k`. Those are the K classes with the highest assigned score by the model.

The last step is to apply the signature calling the `exporter.Exporter.init` method and run the export with the `export` method, which receives an output path, a version number for the model and the session.

```
    scores, class_ids = tf.nn.top_k(y, NUM_CLASSES_TO_RETURN)

    # for simplification we will just return the class ids, we should return
the names instead
    classes = tf.contrib.lookup.index_to_string(tf.to_int64(class_ids),
        mapping=tf.constant([str(i) for i in range(1001)]))

    model_exporter = exporter.Exporter(saver)
    signature = exporter.classification_signature(
        input_tensor=external_x, classes_tensor=classes, scores_ten-
sor=scores)
    model_exporter.init(default_graph_signature=signature, init_op=tf.initi-
alize_all_tables())
    model_exporter.export(sys.argv[1] + "/export", tf.constant(time.time()),
sess)
```

Because of dependencies to auto-generated code in the Exporter class code, you will need to run our exporter using bazel, inside the Docker container.

To do so we will save our code as `export.py` iside the bazel workspace we started before. Will need to a `BUILD` file with a rule for building it like:

```
# BUILD file
```

```
py_binary(
    name = "export",
    srcs = [
        "export.py",
    ],
    deps = [
        "//tensorflow_serving/session_bundle:exporter",
       "@org_tensorflow//tensorflow:tensorflow_py",
        # only needed for inception model export
        "@inception_model//inception",
    ],
)
```

We can then run the exporter from between the container with the command:

```
# on the docker container
cd /mnt/home/serving_example
bazel run :export /tmp/inception-v3
```

And it will create the export in `/tmp/inception-v3/{current_timestamp}/` based on the checkpoint that should be extracted in `/tmp/inception-v3`.

Note that the first time you run it will take some time, because it must compile Tensorflow.

Defining a server interface

The next step is to create a server for the exported model.

Tensorflow Serving is designed to work with gRPC, a binary RPC protocol that works over HTTP/2. It supports a variety of lenguages for creating servers and auto-generating client stubs. As Tensorlow works over C++, we will need to define our server in it. Luckily the server code will be short.

In order to use gRPC, we must define our service contract in a *protocol buffer*, which is the IDL and binary encoding used for gRPC. So lets define our service. As we mentioned in the exporting section, we want our service to have a method that inputs a JPEG encoded string of the image to classify and returns a list of inferred classes with their corresponding scores.

Such a service should be defined in a `classification_service.proto` file like:

```
syntax = "proto3";

message ClassificationRequest {
    // JPEG encoded string of the image.
    bytes input = 1;
};

message ClassificationResponse {
    repeated ClassificationClass classes = 1;
};

message ClassificationClass {
    string name = 1;
    float score = 2;
}

service ClassificationService {
    rpc classify(ClassificationRequest) returns (ClassificationResponse);
}
```

You can use this same interface definition for any kind of service that receives an image, or an audio fragment, or a piece of text.

For using an structured input like a database record, just change the `Classification-Request` message. For example, if we were trying to build the classification service for the Iris dataset:

```
message ClassificationRequest {
    float petalWidth = 1;
    float petalHeight = 2;
    float sepalWidth = 3;
    float sepalHeight = 4;
}
```

The proto file will be converted to the corresponding classes definitions for the client and the server by the proto compiler. To use the protobuf compiler, we have to add a new rule to the `BUILD` file like:

```
load("@protobuf//:protobuf.bzl", "cc_proto_library")

cc_proto_library(
    name="classification_service_proto",
    srcs=["classification_service.proto"],
    cc_libs = ["@protobuf//:protobuf"],
    protoc="@protobuf//:protoc",
    default_runtime="@protobuf//:protobuf",
    use_grpc_plugin=1
)
```

Notice the `load` at the top of the code fragment. It imports the `cc_proto_library` rule definition from the exernally imported protobuf library. Then we use it for defining a build to our proto file. Let's run the build using `bazel build :classification_service_proto` and check the resulting `bazel-genfiles/classification_service.grpc.pb.h`:

```
...

class ClassificationService {
  ...

  class Service : public ::grpc::Service {
     public:
      Service();
      virtual ~Service();
      virtual ::grpc::Status classify(::grpc::ServerContext* context,
const ::ClassificationRequest* request, ::ClassificationResponse* response);
    };
```

`ClassificationService::Service` is the interface we have to implement with the inference logic. We can also check `bazel-genfiles/classification_service.pb.h` for the definitions of the `request` and `response` messages:

```
...

class ClassificationRequest : public ::google::protobuf::Message {
    ...
    const ::std::string& input() const;
    void set_input(const ::std::string& value);
    ...
}

class ClassificationResponse : public ::google::protobuf::Message {
    ...
    const ::ClassificationClass& classes() const;
    void set_allocated_classes(::ClassificationClass* classes);
    ...
}

class ClassificationClass : public ::google::protobuf::Message {
    ...
    const ::std::string& name() const;
    void set_name(const ::std::string& value);
    float score() const;
    void set_score(float value);
    ...
}
```

You can see how the proto definition became a C++ class interface for each type. Their implementations are autogenerated too so we can just use them right away.

Implementing an inference server

To implement `ClassificationService::Service` we will need to load our model export and call inference on it. We do that by the means of a `SessionBundle`, an object that we create from the export and contains a TF session with the fully loaded graph, as well as the metadata including the classification signature defined on the export tool.

To create a SessionBundle from the exported file path, we can define a handy function that handles the boilerplate:

```
#include <iostream>
#include <memory>
#include <string>
```

```
#include <grpc++/grpc++.h>

#include "classification_service.grpc.pb.h"

#include "tensorflow_serving/servables/tensorflow/session_bundle_factory.h"

using namespace std;
using namespace tensorflow::serving;
using namespace grpc;

unique_ptr<SessionBundle> createSessionBundle(const string& pathToExport-
Files) {
        SessionBundleConfig session_bundle_config = SessionBundleConfig();
        unique_ptr<SessionBundleFactory> bundle_factory;
    SessionBundleFactory::Create(session_bundle_config, &bundle_factory);

        unique_ptr<SessionBundle> sessionBundle;
        bundle_factory->CreateSessionBundle(pathToExportFiles, &sessionBun-
dle);

        return sessionBundle;
}
```

In the code we use a `SessionBundleFactory` to create the `SessionBundle` configured to load the model exported in the path specified by `pathToExportFiles`. It returns a unique pointer to the instance of the created `SessionBudle`.

We now have to define the implementation of the service, `ClassificationServiceImpl` that will receive the `SessionBundle` as parameter to be used to do the inference.

```
class ClassificationServiceImpl final : public ClassificationService::Ser-
vice {

  private:
        unique_ptr<SessionBundle> sessionBundle;
  public:
    ClassificationServiceImpl(unique_ptr<SessionBundle> sessionBundle) :
        sessionBundle(move(sessionBundle)) {};
    Status classify(ServerContext* context, const ClassificationRequest* re-
quest,
                    ClassificationResponse* response) override {

                // Load classification signature
                ClassificationSignature signature;
                const tensorflow::Status signatureStatus =
                  GetClassificationSignature(sessionBundle->meta_graph_def,
&signature);

                if (!signatureStatus.ok()) {
                        return Status(StatusCode::INTERNAL, signatureSta-
tus.error_message());
                }

                // Transform protobuf input to inference input tensor.
                tensorflow::Tensor input(tensorflow::DT_STRING, tensor-
flow::TensorShape());
                input.scalar<string>()() = request->input();

                vector<tensorflow::Tensor> outputs;

                // Run inference.
                const tensorflow::Status inferenceStatus = sessionBundle-
>session->Run(
                        {{signature.input().tensor_name(), input}},
                        {signature.classes().tensor_name(), signa-
ture.scores().tensor_name()},
                        {},
                        &outputs);

                if (!inferenceStatus.ok()) {
```

```
                return Status(StatusCode::INTERNAL, inferenceSta-
tus.error_message());
                }

                // Transform inference output tensor to protobuf output.
                for (int i = 0; i < outputs[0].vec<string>().size(); ++i) {
                        ClassificationClass *classificationClass = response-
>add_classes();
                        classificationClass->set_name(out-
puts[0].flat<string>()(i));
                        classificationClass->set_score(out-
puts[1].flat<float>()(i));
                }
        return Status::OK;
    }
};
```

The implementation of the `classify` method does four steps:

- Loads the `ClassificationSignature` stored in the model export meta by using the `GetClassificationSignature` function. The signature specifies the name of the input tensor where to set the received image, and the names of the output tensors in the graph where to obtain the inference results from.
- Copies the JPEG encoded image string from the `request` parameter into a tensor to be sent to inference.
- Runs inference. It obtains the TF session from `sessionBundle` and runs one step on it, passing references to the input and outputs tensors.
- Copies and formats the results from the output tensors to the `response` output param in the shape specified by the `ClassificationResponse` message.

The last piece of code is the boilerplate to setup a gRPC `Server` and create an instance of our `ClassificationServiceImpl`, configured with the `SessionBundle`.

```
int main(int argc, char** argv) {

    if (argc < 3) {
     cerr << "Usage: server <port> /path/to/export/files" << endl;
                return 1;
    }

        const string serverAddress(string("0.0.0.0:") + argv[1]);
        const string pathToExportFiles(argv[2]);

        unique_ptr<SessionBundle> sessionBundle = createSessionBundle(path-
ToExportFiles);

        ClassificationServiceImpl classificationServiceImpl(move(sessionBun-
dle));

    ServerBuilder builder;
    builder.AddListeningPort(serverAddress, grpc::InsecureServerCreden-
tials());
    builder.RegisterService(&classificationServiceImpl);

    unique_ptr<Server> server = builder.BuildAndStart();
    cout << "Server listening on " << serverAddress << endl;

    server->Wait();

    return 0;
}
```

To compile this code we have to define a rule in our `BUILD` file for it

```
cc_binary(
    name = "server",
    srcs = [
        "server.cc",
        ],
```

```
    deps = [
        ":classification_service_proto",
        "@tf_serving//tensorflow_serving/servables/tensor-
flow:session_bundle_factory",
        "@grpc//:grpc++",
        ],
)
```

With this code we can run the inference server from the container with `bazel run :server 9999 /tmp/inception-v3/export/{timestamp}`.

The client app

As gRPC works over HTTP/2, it may allow in the future to call gRPC based services directly from the browser. But until the manistream of browsers support the required HTTP/2 features and Google releases a browser side Javascript gRPC client, accessing our inference service from a webapp should be done through a server side component.

We are going to do then a really simple Python web server based on `BaseHTTPServer` that will handle the image file upload and send for processing to inference, returning the inference result in plain text.

Our server will respond `GET` requests with a simple form for sending the image to classify. The code for it:

```
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

import cgi
import classification_service_pb2
from grpc.beta import implementations

class ClientApp(BaseHTTPRequestHandler):
    def do_GET(self):
        self.respond_form()

    def respond_form(self, response=""):

        form = """
        <html><body>
        <h1>Image classification service</h1>
        <form enctype="multipart/form-data" method="post">
        <div>Image: <input type="file" name="file" accept="image/jpeg"></div>
        <div><input type="submit" value="Upload"></div>
        </form>
        %s
        </body></html>
        """

        response = form % response

        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-length", len(response))
        self.end_headers()
        self.wfile.write(response)
```

To call inference from our webapp server, we need the corresponding Python protocol buffer client for the `ClassificationService`. To generate it we will need to run the protocol buffer compiler for Python:

```
pip install grpcio cython grpcio-tools
python -m grpc.tools.protoc -I. --python_out=. --grpc_python_out=. classifi-
cation_service.proto
```

It will generate the `classification_service_pb2.py` file that contains the `stub` for calling the service.

On `POST` the server will parse the sent form and create a ClassificationRequest with it. Then setup a `channel` to the classification server and submit the request to it. Finally, it will render the classification response as HTML and send it back to the user.

```
    def do_POST(self):

        form = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={
                'REQUEST_METHOD': 'POST',
```

```
                'CONTENT_TYPE': self.headers['Content-Type'],
            })

        request = classification_service_pb2.ClassificationRequest()
        request.input = form['file'].file.read()

        channel = implementations.insecure_channel("127.0.0.1", 9999)
        stub = classification_service_pb2.beta_create_ClassificationSer-
vice_stub(channel)
        response = stub.classify(request, 10) # 10 secs timeout

        self.respond_form("<div>Response: %s</div>" % response)
```

To run the server we can `python client.py` from outside the container. Then we navigate with a browser to http://localhost:8080 to access its UI. Go ahead and upload an image to try inference working on it.

Preparing for production

To close the chapter we will learn how to put our classification server in production.

We start by copying the compiled server files to a permanent location inside the container, and cleaning up all the temporary build files:

```
# inside the container
mkdir /opt/classification_server
cd /mnt/home/serving_example
cp -R bazel-bin/. /opt/classification_server
bazel clean
```

Now, outside the container we have to commit its state into a new Docker image. That basically means creating a snapshot of the changes in its virtual file system.

```
# outside the container
docker ps

# grab container id from above
docker commit <container id>
```

That's it. Now we can push the image to our favorite docker serving cloud and start serving it.

Conclusion ~~~~~~~~

In this chapter we learned how to adapt our models for serving, exporting them and building fast lightweight servers that run them. We also learned how to create simple web apps for consuming them giving the full toolset for consuming Tensorflow models from other apps.

In the next chapter we provide code snippets and explanations for some of the helper functions and classes used throughout this book.