Chapter 8. Model Deployment with TensorFlow Serving

The deployment of your machine learning model is the last step before others can use your model and make predictions with it. Unfortunately, the deployment of machine learning models falls into a gray zone in today's thinking of the division of labor in the digital world. It isn't just a DevOps task since it requires some knowledge of the model architecture and its hardware requirements. At the same time, deploying machine learning models is a bit outside the comfort zone of machine learning engineers and data scientists. They know their models inside out but tend to struggle with the deployment of machine learning models. In this and the following chapter, we want to bridge the gap between the worlds and guide data scientists and DevOps engineers through the steps to deploy machine learning models. Figure 8-1 shows the position of the deployment step in a machine learning pipeline.

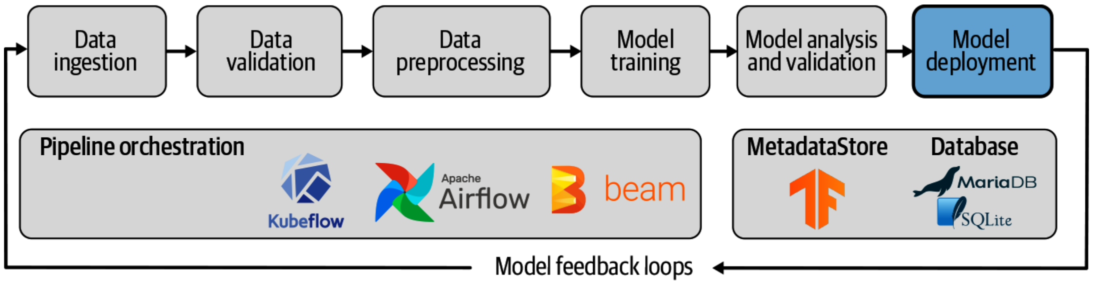


Figure 8-1. Model deployments as part of ML pipelines

Machine learning models can be deployed in three main ways: with a model server, in a user's browser, or on an edge device. The most common way today to deploy a machine learning model is with a model server, which we will focus on in this chapter. The client that requests a prediction submits the input data to the model server and in return receives a prediction. This requires that the client can connect with the model server.

There are situations when you don't want to submit the input data to a model server (e.g., when the input data is sensitive, or when there are privacy concerns). In these situations, you can deploy the machine learning model to a user's browser. For example, if you want to determine whether an image contains sensitive information, you could classify the sensitivity level of the image before it is uploaded to a cloud server.

However, there is also a third type of model deployment: deploying to edge devices. There are situations that don't allow you to connect to a model server to make predictions (i.e., remote sensors or IoT devices). The number of applications being deployed to edge devices is increasing, making it a valid option for model deployments. In Chapter 10, we discuss how TensorFlow models can be converted to TFLite models, which can be executed on edge devices.

In this chapter, we highlight TensorFlow's Serving module, a simple and consistent way of deploying TensorFlow models through a model server. We will introduce its setup and discuss efficient deployment options. This is not the only way of deploying deep learning models; there are a few alternative options, which we discuss toward the end of this chapter.

Let's start the chapter with how you shouldn't set up a model server before we take a deep dive into TensorFlow Serving.

A Simple Model Server

Most introductions to deploying machine learning models follow roughly the same workflow:

- Create a web app with Python (i.e., with web frameworks like Flask or Django).
- Create an API endpoint in the web app, as we show in Example 8-1.
- Load the model structure and its weights.
- Call the predict method on the loaded model.
- Return the prediction results as an HTTP request.

Example 8-1. Example setup of a Flask endpoint to infer model predictions

```
import json
from flask import Flask, request
from tensorflow.keras.models import load_model
from utils import preprocess ❶

model = load_model('model.h5') ❷
app = Flask(__name__)

@app.route('/classify', methods=['POST'])
def classify():
    complaint_data = request.form["complaint_data"]
    preprocessed_complaint_data = preprocess(complaint_data)
    prediction = model.predict([preprocessed_complaint_data]) ❸
    return json.dumps({"score": prediction}) ❹
```

❶ Preprocessing to convert data structure.

❷ Load your trained model.

❸ Perform the prediction.

❹ Return the prediction in an HTTP response.

This setup is a quick and easy implementation, perfect for demonstration projects. However, we do not recommend using Example 8-1 to deploy machine learning models to production endpoints.

Next, let's discuss why we don't recommend deploying machine learning models with such a setup. The reason is our benchmark for our proposed deployment solution.

The Downside of Model Deployments with Python-Based APIs

While the Example 8-1 implementation can be sufficient for demonstration purposes, such deployments often face challenges. The challenges start with proper separation between the API and the data science code, a consistent API structure and the resulting inconsistent model versioning, and inefficient model inferences. We will take a closer look at these challenges in the following sections.

Lack of Code Separation

In Example 8-1, we assumed that the trained model was being deployed with the same API code base that it was also living in. This means that there would be no separation between the API code and the machine learning model, which can be problematic when data scientists want to update a model and such an update requires coordination with the API team. Such coordination also requires that the API and data science teams work in sync to avoid unnecessary delays on the model deployments.

An intertwined API and data science code base also creates ambiguity around API ownership.

The lack of code separation also requires that the model has to be loaded in the same programming language as the API code. This mixing of backend and data science code can ultimately prevent your API team from upgrading your API backend. However, it also provides a good separation of responsibilities: the data scientists can focus on model training and the DevOps colleagues can focus on the deployment of the trained models.

We highlight how you can separate your models from your API code effectively and simplify your deployment workflows in "TensorFlow Serving".

Lack of Model Version Control

Example 8-1 doesn't make any provision for different model versions. If you wanted to add a new version, you would have to create a new endpoint (or add some branching logic to the existing endpoint). This requires extra attention to keep all endpoints structurally the same, and it requires a lot of boilerplate code.

The lack of model version control also requires the API and the data science teams to coordinate which version is the default version and how to phase in new models.

Inefficient Model Inference

For any request to your prediction endpoint written in the Flask setup as shown in Example 8-1, a full round trip is performed. This means each request is preprocessed and inferred individually. The key reason why we argue that such a setup is only for demonstration purposes is that it is highly inefficient. During the training of your model, you will probably use a batching technique that allows you to compute multiple samples at the same time and then apply the gradient change for your batch to your network's weights. You can apply the same technique when you want the model to make predictions. A model server can gather all requests during an acceptable timeframe or until the batch is full and ask the model for its predictions. This is an especially effective method when the inference runs on GPUs.

In "Batching Inference Requests", we introduce how you can easily set up such a batching behavior for your model server.

TensorFlow Serving

As you have seen in the earlier chapters of this book, TensorFlow comes with a fantastic ecosystem of extensions and tools. One of the earlier open source extensions was TensorFlow Serving. It allows you to deploy any TensorFlow graph, and you can make predictions from the graph through its standardized endpoints. As we discuss in a moment, TensorFlow Serving handles the model and version management for you, lets you serve models based on policies, and allows you to load your models from various sources. At the same time, it is focused on high-performance throughput for low-latency predictions. TensorFlow Serving is used internally at Google and has been adopted by a good number of corporations and startups.[1]

TensorFlow Architecture Overview

TensorFlow Serving provides you the functionality to load models from a given source (e.g., AWS S3 buckets) and notifies the *loader* if the source has changed. As Figure 8-2 shows, everything behind the scenes of TensorFlow Serving is controlled by a model manager, which manages when to update the models and which model is used for the predictions.

The rules for the inference determination are set by the policy which is managed by the model manager. Depending on your configuration, you can, for example, load one model at a time and have the model update automatically once the source module detects a newer version.

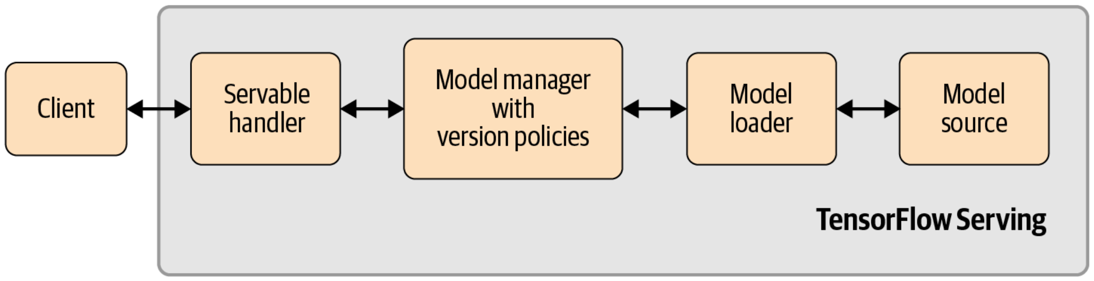


Figure 8-2. Overview of the TensorFlow Serving architecture

Exporting Models for TensorFlow Serving

Before we dive into the TensorFlow Serving configurations, let's discuss how you can export your machine learning models so that they can be used by TensorFlow Serving.

Depending on your type of TensorFlow model, the export steps are slightly different. The exported models have the same file structure as we see in Example 8-2. For Keras models, you can use:

```
saved_model_path = model.save(file path="./saved_models", save_format="tf")
```

ADD A TIMESTAMP TO YOUR EXPORT PATH

It is recommended to add the timestamp of the export time to the export path for the Keras model when you are manually saving the model. Unlike the save method of `tf.Estimator`, `model.save()` doesn't create the timestamped path automatically. You can create the file path easily with the following Python code:

```
import time

ts = int(time.time())
file path = "./saved_models/{}".format(ts)
saved_model_path = model.save(file path=file path,
                              save_format="tf")
```

For TensorFlow Estimator models, you need to first declare a `receiver function`:

```
import tensorflow as tf

def serving_input_receiver_fn():
    # an example input feature
    input_feature = tf.compat.v1.placeholder(
        dtype=tf.string, shape=[None, 1], name="input")

    fn = tf.estimator.export.build_raw_serving_input_receiver_fn(
        features={"input_feature": input_feature})
    return fn
```

Export the Estimator model with the `export_saved_model` method of the `Estimator`:

```
estimator = tf.estimator.Estimator(model_fn, "model", params={})
estimator.export_saved_model(
    export_dir_base="saved_models/",
    serving_input_receiver_fn=serving_input_receiver_fn)
```

Both export methods produce output which looks similar to the following example:

```
...
INFO:tensorflow:Signatures INCLUDED in export for Classify: None
INFO:tensorflow:Signatures INCLUDED in export for Regress: None
INFO:tensorflow:Signatures INCLUDED in export for Predict: ['serving_default']
INFO:tensorflow:Signatures INCLUDED in export for Train: None
INFO:tensorflow:Signatures INCLUDED in export for Eval: None
INFO:tensorflow:No assets to save.
INFO:tensorflow:No assets to write.
INFO:tensorflow:SavedModel written to: saved_models/1555875926/saved_model.pb
Model exported to:  b'saved_models/1555875926'
```

In our model export examples, we specified the folder *saved_models/* as the model destination. For every exported model, TensorFlow creates a directory with the timestamp of the export as its folder name:

Example 8-2. Folder and file structure of exported models

```
$ tree saved_models/
saved_models/
└── 1555875926
    ├── assets
    │   └── saved_model.json
    ├── saved_model.pb
    └── variables
        ├── checkpoint
        ├── variables.data-00000-of-00001
        └── variables.index

3 directories, 5 files
```

The folder contains the following files and subdirectories:

saved_model.pb

The binary protocol buffer file contains the exported model graph structure as a `MetaGraphDef` object.

variables

The folder contains the binary files with the exported variable values and checkpoints corresponding to the exported model graph.

assets

This folder is created when additional files are needed to load the exported model. The additional file can include vocabularies, which saw in Chapter 5.

Model Signatures

Model signatures identify the model graph's inputs and outputs as well as the *method* of the graph signature. The definition of the input and output signatures allows us to map serving inputs to a given graph node for the inference. These mappings are useful if we want to update the model without changing the requests to the model server.

In addition, the *method* of the model defines an expected pattern of inputs and outputs. At the moment, there are three supported signature types: predict, classify, or regress. We will take a closer look at the details in the following section.

Signature Methods

The most flexible signature method is *predict*. If we don't specify a different signature method, TensorFlow will use *predict* as the default method. Example 8-3 shows an example signature for the method *predict*. In the example, we are mapping the key `inputs` to the graph node with the name *sentence*. The prediction from the model is the output of the graph node *y*, which we are mapping to the output key `scores`.

The *predict* method allows you to define additional output nodes. It is useful to add more inference outputs when you want to capture the output of an attention layer for visualizations or to debug a network node.

Example 8-3. Example model prediction signature

```
signature_def: {
  key  : "prediction_signature"
  value: {
    inputs: {
      key  : "inputs"
```

```
        value: {
          name: "sentence:0"
          dtype: DT_STRING
          tensor_shape: ...
        },
        ...
      }
      outputs: {
        key  : "scores"
        value: {
          name: "y:0"
          dtype: ...
          tensor_shape: ...
        }
      }
      method_name: "tensorflow/serving/predict"
    }
  }
```

Another signature method is *classify*. The method expects one input with the name *inputs* and provides two output tensors, *classes* and *scores*. At least one of the output tensors needs to be defined. In our example shown in Example 8-4, a classification model takes the input `sentence` and outputs the predicted `classes` together with the corresponding `scores`.

Example 8-4. Example model classification signature

```
  signature_def: {
    key  : "classification_signature"
    value: {
      inputs: {
        key  : "inputs"
        value: {
          name: "sentence:0"
          dtype: DT_STRING
          tensor_shape: ...
        }
      }
      outputs: {
        key  : "classes"
        value: {
          name: "y_classes:0"
          dtype: DT_UINT16
          tensor_shape: ...
        }
      }
      outputs: {
        key   : "scores"
        value: {
          name: "y:0"
          dtype: DT_FLOAT
          tensor_shape: ...
        }
      }
      method_name: "tensorflow/serving/classify"
    }
  }
```

The third available signature method is *regress*. This method takes only one input named *inputs* and provides only output with the name *outputs*. This signature method is designed for regression models. Example 8-5 shows an example of a *regress* signature.

Example 8-5. Example model regression signature

```
signature_def: {
  key   : "regression_signature"
  value: {
    inputs: {
      key   : "inputs"
      value: {
        name: "input_tensor_0"
        dtype: ...
        tensor_shape: ...
      }
    }
    outputs: {
      key   : "outputs"
      value: {
        name: "y_outputs_0"
        dtype: DT_FLOAT
        tensor_shape: ...
      }
    }
    method_name: "tensorflow/serving/regress"
    }
  }
```

In “URL structure”, we will see the signature methods again when we define the URL structure for our model endpoints.

Inspecting Exported Models

After all the talk about exporting your model and the corresponding model signatures, let’s discuss how you can inspect the exported models before deploying them with TensorFlow Serving.

You can install the TensorFlow Serving Python API with the following `pip` command:

```
$ pip install tensorflow-serving-api
```

After the installation, you have access to a useful command-line tool called SavedModel Command Line Interface (CLI). This tool lets you:

Inspect the signatures of exported models

This is very useful primarily when you don’t export the model yourself, and you want to learn about the inputs and outputs of the model graph.

Test the exported models

The CLI tools let you infer the model without deploying it with TensorFlow Serving. This is extremely useful when you want to test your model input data.

We’ll cover both use cases in the following two sections.

Inspecting the Model

`saved_model_cli` helps you understand the model dependencies without inspecting the original graph code.

If you don't know the available tag-sets,[2] you can inspect the model with:

```
$ saved_model_cli show --dir saved_models/
The given SavedModel contains the following tag-sets:
serve
```

If your model contains different graphs for different environments (e.g., a graph for a CPU or GPU inference), you will see multiple tags. If your model contains multiple tags, you need to specify a tag to inspect the details of the model.

Once you know the `tag_set` you want to inspect, add it as an argument, and `saved_model_cli` will provide you the available model signatures. Our demo model has only one signature, which is called `serving_default`:

```
$ saved_model_cli show --dir saved_models/ --tag_set serve
The given SavedModel 'MetaGraphDef' contains 'SignatureDefs' with the
following keys:
SignatureDef key: "serving_default"
```

With the `tag_set` and `signature_def` information, you can now inspect the model's inputs and outputs. To obtain the detailed information, add the `signature_def` to the CLI arguments.

The following example signature is taken from our model that was produced by our demonstration pipeline. In Example 6-4, we defined our signature function, which takes serialized `tf.Example` records as inputs and provides the prediction through the output Tensor *outputs*, as shown in the following model signature:

```
$ saved_model_cli show --dir saved_models/ \
        --tag_set serve --signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['examples'] tensor_info:
      dtype: DT_STRING
      shape: (-1)
      name: serving_default_examples:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['outputs'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: StatefulPartitionedCall_1:0
Method name is: tensorflow/serving/predict
```

If you want to see all signatures regardless of the `tag_set` and `signature_def`, you can use the `--all` argument:

```
$ saved_model_cli show --dir saved_models/ --all
...
```

After we investigated the model's signature, we can now test the model inference before we deploy the machine learning model.

Testing the Model

`saved_model_cli` also lets you test the export model with sample input data.

You have three different ways to submit the sample input data for the model test inference:

`--inputs`

The argument points at a NumPy file containing the input data formatted as NumPy `ndarray`.

`--input_exprs`

The argument allows you to define a Python expression to specify the input data. You can use NumPy functionality in your expressions.

`--input_examples`

The argument is expecting the input data formatted as a `tf.Example` data structure (see Chapter 4).

For testing the model, you can specify exactly one of the input arguments. Furthermore, `saved_model_cli` provides three optional arguments:

--outdir

`saved_model_cli` will write any graph output to `stdout`. If you would rather write the output to a file, you can specify the target directory with `--outdir`.

--overwrite

If you opt for writing the output to a file, you can specify with `--overwrite` that the files can be overwritten.

--tf_debug

If you want to further inspect the model, you can step through the model graph with the TensorFlow Debugger (TFDBG).

```
$ saved_model_cli run --dir saved_models/ \
                      --tag_set serve \
                      --signature_def x1_x2_to_y \
                      --input_examples 'examples=[{"company": "HSBC", ...}]'
```

After all the introduction of how to export and inspect models, let's dive into the TensorFlow Serving installation, setup, and operation.

Setting Up TensorFlow Serving

There are two easy ways to get TensorFlow Serving installed on your serving instances. You can either run TensorFlow Serving on Docker or, if you run an Ubuntu OS on your serving instances, you can install the Ubuntu package.

Docker Installation

The easiest way of installing TensorFlow Serving is to download the pre-built Docker image.[3] As you have seen in Chapter 2, you can obtain the image by running:

```
$ docker pull tensorflow/serving
```

If you are running the Docker container on an instance with GPUs available, you will need to download the latest build with GPU support.

```
$ docker pull tensorflow/serving:latest-gpu
```

The Docker image with GPU support requires Nvidia's Docker support for GPUs. The installation steps can be found on the company's website.

Native Ubuntu Installation

If you want to run TensorFlow Serving without the overhead of running Docker, you can install Linux binary packages available for Ubuntu distributions.

The installation steps are similar to other nonstandard Ubuntu packages. First, you need to add a new package source to the distribution's source list or add a new list file to the `sources.list.d` directory by executing the following in your Linux terminal:

```
$ echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-serving-apt \
  stable tensorflow-model-server tensorflow-model-server-universal" \
  | sudo tee /etc/apt/sources.list.d/tensorflow-serving.list
```

Before updating your package registry, you should add the packages' public key to your distribution's key chain:

```
$ curl https://storage.googleapis.com/tensorflow-serving-apt/\
tensorflow-serving.release.pub.gpg | sudo apt-key add -
```

After updating your package registry, you can install TensorFlow Serving on your Ubuntu operating system:

```
$ apt-get update
$ apt-get install tensorflow-model-server
```

TWO UBUNTU PACKAGES FOR TENSORFLOW SERVING

Google provides two Ubuntu packages for TensorFlow Serving! The earlier referenced `tensorflow-model-server` package is the preferred package, and it comes with specific CPU optimizations precompiled (e.g., AVX instructions).

At the time of writing this chapter, a second package with the name `tensorflow-model-server-universal` is also provided. It doesn't contain the precompiled optimizations and can, therefore, be run on old hardware (e.g., CPUs without the AVX instruction set).

Building TensorFlow Serving from Source

It is recommended to run TensorFlow Serving with the prebuilt Docker image or Ubuntu packages. In some situations, you have to compile TensorFlow Serving, for example when you want to optimize the model serving for your underlying hardware. At the moment, you can only build TensorFlow Serving for Linux operating systems, and the build tool `bazel` is required. You can find detailed instructions in the TensorFlow Serving documentation.

OPTIMIZE YOUR TENSORFLOW SERVING INSTANCES

If you build TensorFlow Serving from scratch, we highly recommend compiling the Serving version for the specific TensorFlow version of your models and available hardware of your serving instances.

Configuring a TensorFlow Server

Out of the box, TensorFlow Serving can run in two different modes. First, you can specify a model, and have TensorFlow Serving always provide the latest model. Alternatively, you can specify a configuration file with all models and versions that you want to load, and have TensorFlow Serving load all the named models.

Single Model Configuration

If you want to run TensorFlow Serving by loading a single model and switching to newer model versions when they are available, the single model configuration is preferred.

If you run TensorFlow Serving in a Docker environment, you can run the `tensorflow\serving` image with the following command:

```
$ docker run -p 8500:8500 \ ❶
             -p 8501:8501 \
             --mount type=bind,source=/tmp/models,target=/models/my_model \ ❷
             -e MODEL_NAME=my_model \ ❸
             -e MODEL_BASE_PATH=/models/my_model \
             -t tensorflow/serving ❹
```

❶ Specify the default ports.

❷ Mount the model directory.

❸ Specify your model.

❹ Specify the docker image.

By default, TensorFlow Serving is configured to create a representational state transfer (REST) and Google Remote Procedure Calls (gRPC) endpoint. By specifying both ports, 8500 and 8501, we expose the REST and gRPC capabilities.[4] The docker `run` command creates a mount between a folder on the host (source) and the container (target) filesystem. In Chapter 2, we discussed how to pass environment variables to the docker container. To run the server in a single model configuration, you need to specify the model name `MODEL_NAME`.

If you want to run the Docker image prebuilt for GPU images, you need to swap out the name of the docker image to the latest GPU build with:

```
$ docker run ...
             -t tensorflow/serving:latest-gpu
```

If you have decided to run TensorFlow Serving without the Docker container, you can run it with the command:

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --model_name=my_model \
                          --model_base_path=/models/my_model
```

In both scenarios, you should see output on your terminal that is similar to the following:

```
2019-04-26 03:51:20.304826: I
tensorflow_serving/model_servers/
server.cc:82]
  Building single TensorFlow model file config:
  model_name: my_model model_base_path: /models/my_model
2019-04-26 03:51:20: I tensorflow_serving/model_servers/server_core.cc:461]
  Adding/updating models.
2019-04-26 03:51:20: I
tensorflow_serving/model_servers/
server_core.cc:558]
  (Re-)adding model: my_model
...
2019-04-26 03:51:34.507436: I tensorflow_serving/core/loader_harness.cc:86]
  Successfully loaded servable version {name: my_model version: 1556250435}
2019-04-26 03:51:34.516601: I tensorflow_serving/model_servers/server.cc:313]
  Running gRPC ModelServer at 0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename not supported
[evhttp_server.cc : 237] RAW: Entering the event loop ...
2019-04-26 03:51:34.520287: I tensorflow_serving/model_servers/server.cc:333]
  Exporting HTTP/REST API at:localhost:8501 ...
```

From the server output, you can see that the server loaded our model `my_model` successfully, and that created two endpoints: one REST and one gRPC endpoint.

TensorFlow Serving makes the deployment of machine learning models extremely easy. One great advantage of serving models with TensorFlow Serving is the *hot swap* capability. If a new model is uploaded, the server's model manager will detect the new version, unload the existing model, and load the newer model for inferencing.

Let's say you update the model and export the new model version to the mounted folder on the host machine (if you are running with the docker setup) and no configuration change is required. The model manager will detect the newer model and reload the endpoints. It will notify you about the unloading of the older model and the loading of the newer model. In your terminal, you should find messages like:

```
2019-04-30 00:21:56.486988: I tensorflow_serving/core/basic_manager.cc:739]
  Successfully reserved resources to load servable
  {name: my_model version: 1556583584}
2019-04-30 00:21:56.487043: I tensorflow_serving/core/loader_harness.cc:66]
  Approving load for servable version {name: my_model version: 1556583584}
2019-04-30 00:21:56.487071: I tensorflow_serving/core/loader_harness.cc:74]
  Loading servable version {name: my_model version: 1556583584}
...
2019-04-30 00:22:08.839375: I tensorflow_serving/core/loader_harness.cc:119]
  Unloading servable version {name: my_model version: 1556583236}
2019-04-30 00:22:10.292695: I ./tensorflow_serving/core/simple_loader.h:294]
  Calling MallocExtension_ReleaseToSystem() after servable unload with 12623389
2019-04-30 00:22:10.292771: I tensorflow_serving/core/loader_harness.cc:127]
  Done unloading servable version {name: my_model version: 1556583236}
```

By default, TensorFlow Serving will load the model with the highest version number. If you use the export methods shown earlier in this chapter, all models will be exported in folders with the epoch timestamp as the folder name. Therefore, newer models will have a higher version number than older models.

The same default model loading policy of TensorFlow Serving also allows model rollbacks. In case you want to roll back a model version, you can delete the model version from the base path. The model server will then detect the removal of the version with the next polling of the filesystem,[5] unload the deleted model, and load the most recent, existing model version.

Multiple Model Configuration

You can also configure TensorFlow Serving to load multiple models at the same time. To do that, you need to create a configuration file to specify the models:

```
model_config_list {
  config {
    name: 'my_model'
    base_path: '/models/my_model/'
    model_platform: 'tensorflow'
  }
  config {
    name: 'another_model'
    base_path: '/models/another_model/'
    model_platform: 'tensorflow'
  }
}
```

The configuration file contains one or more `config` dictionaries, all listed below a `model_config_list` key.

In your Docker configuration, you can mount the configuration file and load the model server with the configuration file instead of a single model:

```
$ docker run -p 8500:8500 \
             -p 8501:8501 \
             --mount type=bind,source=/tmp/models,target=/models/my_model \
             --mount type=bind,source=/tmp/model_config,\
             target=/models/model_config \ ❶
             -e MODEL_NAME=my_model \
             -t tensorflow/serving \
             --model_config_file=/models/model_config ❷
```

❶ Mount the configuration file.

❷ Specify the model configuration file.

If you use TensorFlow Serving outside of a Docker container, you can point the model server to the configuration file with the argument `model_config_file`, which loads and the configuration from the file:

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --model_config_file=/models/model_config
```

CONFIGURE SPECIFIC MODEL VERSIONS

There are situations when you want to load not just the latest model version, but either all or specific model versions. For example, you may want to do model A/B testing, as we will discuss in "Model A/B Testing with TensorFlow Serving", or provide a stable and a development model version. TensorFlow Serving, by default, always loads the latest model version. If you want to load a set of available model versions, you can extend the model configuration file with:

```
...
config {
  name: 'another_model'
  base_path: '/models/another_model/'
  model_version_policy: {all: {}}
}
...
```

If you want to specify specific model versions, you can define them as well:

```
...
config {
  name: 'another_model'
  base_path: '/models/another_model/'
  model_version_policy {
    specific {
      versions: 1556250435
      versions: 1556251435
    }
  }
}
...
```

You can even give the model version labels. The labels can be extremely handy later when you want to make predictions from the models. At the time of writing, version labels were only available through TensorFlow Serving's gRPC endpoints:

```
...
model_version_policy {
  specific {
    versions: 1556250435
    versions: 1556251435
  }
}
version_labels {
  key: 'stable'
  value: 1556250435
}
version_labels {
  key: 'testing'
  value: 1556251435
}
...
```

With the model version now configured, we can use those endpoints for the versions to run our model A/B test. If you are interested in how to infer these model versions, we recommend “Model A/B Testing with TensorFlow Serving” for an example of a simple implementation.

Starting with TensorFlow Serving 2.3, the *version_label* functionality will be available for REST endpoints in addition to the existing gRPC functionality of TensorFlow Serving.

REST Versus gRPC

In “Single Model Configuration”, we discussed how TensorFlow Serving allows two different API types: REST and gRPC. Both protocols have their advantages and disadvantages, and we would like to take a moment to introduce both before we dive into how you can communicate with these endpoints.

REST

REST is a communication “protocol” used by today’s web services. It isn’t a formal protocol, but more a communication style that defines how clients communicate with web services. REST clients communicate with the server using the standard HTTP methods like `GET`, `POST`, `DELETE`, etc. The payloads of the requests are often encoded as XML or JSON data formats.

gRPC

gRPC is a remote procedure protocol developed by Google. While gRPC supports different data formats, the standard data format used with gRPC is protocol buffer, which we used throughout this book. gRPC provides low-latency communication and smaller payloads if protocol buffers are used. gRPC was designed with APIs in mind. The downside is that the payloads are in a binary format, which can make a quick inspection difficult.

WHICH PROTOCOL TO USE?

On the one hand, it looks very convenient to communicate with the model server over REST. The endpoints are easy to infer, the payloads can be easily inspected, and the endpoints can be tested with `curl` requests or browser tools. REST libraries are widely available for all sorts of clients and often are already available on the client system (i.e., a mobile application).

On the other hand, gRPC APIs have a higher burden of entry initially. gRPC libraries often need to be installed on the client side. However, they can lead to significant performance improvements depending on the data structures required for the model inference. If your model experiences many requests, the reduced payload size due to the protocol buffer serialization can be beneficial.

Internally, TensorFlow Serving converts JSON data structures submitted via REST to `tf.Example` data structures, and this can lead to slower performance. Therefore, you might see better performance with gRPC requests if the conversion requires many type conversions (i.e., if you submit a large array with float values).

Making Predictions from the Model Server

Until now, we have entirely focused on the model server setup. In this section, we want to demonstrate how a client (e.g., a web app), can interact with the model server. All code examples concerning REST or gRPC requests are executed on the client side.

Getting Model Predictions via REST

To call the model server over REST, you'll need a Python library to facilitate the communication for you. The standard library these days is `requests`. Install the library:

```
$ pip install requests
```

The following example showcases an example `POST` request.

```
import requests

url = "http://some-domain.abc"
payload = {"key_1": "value_1"}
r = requests.post(url, json=payload) ❶
print(r.json()) ❷
# {'data': ...}
```

❶ Submit the request.

❷ View the HTTP response.

URL structure

The URL for your HTTP request to the model server contains information about which model and which version you would like to infer:

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}:{VERB}
```

HOST

The host is the IP address or domain name of your model server. If you run your model server on the same machine where you run your client code, you can set the host to `localhost`.

PORT

You'll need to specify the port in your request URL. The standard port for the REST API is 8501. If this conflicts with other services in your service ecosystem, you can change the port in your server arguments during the startup of the server.

MODEL_NAME

The model name needs to match the name of your model when you either set up your model configuration or started up the model server.

VERB

The type of model is specified through the verb in the URL. You have three options: `predict`, `classify`, or `regress`. The verb corresponds to the signature methods of the endpoint.

MODEL_VERSION

If you want to make predictions from a specific model version, you'll need to extend the URL with the model version identifier:

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}[/versions/${MODEL_VERSION}]:{VERB}
```

Payloads

With the URL in place, let's discuss the request payloads. TensorFlow Serving expects the input data as a JSON data structure, as shown in the following example:

```
{
  "signature_name": <string>,
  "instances": <value>
}
```

The `signature_name` is not required. If it isn't specified, the model server will infer the model graph signed with the default `serving` label.

The input data is expected either as a list of objects or as a list of input values. To submit multiple data samples, you can submit them as a list under the `instances` key.

If you want to submit one data example for the inference, you can use `inputs` and list all input values as a list. One of the keys, `instances` and `inputs`, has to be present, but never both at the same time:

```
{
  "signature_name": <string>,
  "inputs": <value>
}
```

Example 8-6 shows an example of how to request a model prediction from our TensorFlow Serving endpoint. We only submit one data example for the inference in our example, but we could easily submit a list of data inputs representing multiple requests.

Example 8-6. Example model prediction request with a Python client

```
import requests

def get_rest_request(text, model_name="my_model"):
    url = "http://localhost:8501/v1/models/{}:predict".format(model_name) ❶
    payload = {"instances": [text]} ❷
    response = requests.post(url=url, json=payload)
    return response

rs_rest = get_rest_request(text="classify my text")
rs_rest.json()
```

❶ Exchange `localhost` with an IP address if the server is not running on the same machine.

❷

Add more examples to the `instance` list if you want to infer more samples.

Using TensorFlow Serving via gRPC

If you want to use the model with gRPC, the steps are slightly different from the REST API requests.

First, you establish a gRPC `channel`. The channel provides the connection to the gRPC server at a given host address and over a given port. If you require a secure connection, you need to establish a secure channel at this point. Once the channel is established, you'll create a `stub`. A `stub` is a local object which replicates the available methods from the server:

```
import grpc
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
import tensorflow as tf

def create_grpc_stub(host, port=8500):
    hostport = "{}:{}".format(host, port)
    channel = grpc.insecure_channel(hostport)
    stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
    return stub
```

Once the gRPC stub is created, we can set the model and the signature to access predictions from the correct model and submit our data for the inference:

```
def grpc_request(stub, data_sample, model_name='my_model', \
                 signature_name='classification'):
    request = predict_pb2.PredictRequest()
    request.model_spec.name = model_name
    request.model_spec.signature_name = signature_name

    request.inputs['inputs'].CopyFrom(tf.make_tensor_proto(data_sample,
                                                    shape=[1,1])) ❶
    result_future = stub.Predict.future(request, 10) ❷
    return result_future
```

❶ `inputs` is the name of the input of our neural network.

❷ 10 is the max time in seconds before the function times out.

With the two function, now available, we can infer our example datasets with the two function calls:

```
stub = create_grpc_stub(host, port=8500)
rs_grpc = grpc_request(stub, data)
```

SECURE CONNECTIONS

The `grpc` library also provides functionality to connect securely with the gRPC endpoints. The following example shows how to create a secure channel with gRPC from the client side:

```
import grpc

cert = open(client_cert_file, 'rb').read()
key = open(client_key_file, 'rb').read()
ca_cert = open(ca_cert_file, 'rb').read() if ca_cert_file else ''
credentials = grpc.ssl_channel_credentials(
    ca_cert, key, cert
)
channel = implementations.secure_channel(hostport, credentials)
```

On the server side, TensorFlow Serving can terminate secure connections if SSL is configured. To terminate secure connections, create an *SSL configuration file* as shown in the following example:[6]

```
server_key:  "-----BEGIN PRIVATE KEY-----\n
              <your_ssl_key>\n
              -----END PRIVATE KEY-----"
server_cert: "-----BEGIN CERTIFICATE-----\n
              <your_ssl_cert>\n
              -----END CERTIFICATE-----"
custom_ca: ""
client_verify: false
```

Once you have created the configuration file, you can pass the file path to the TensorFlow Serving argument `--ssl_config_file` during the start of TensorFlow Serving:

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --model_name=my_model \
                          --model_base_path=/models/my_model \
                          --ssl_config_file="<path_to_config_file>"
```

Getting predictions from classification and regression models

If you are interested in making predictions from classification and regression models, you can use the gRPC API.

If you would like to get predictions from a classification model, you will need to swap out the following lines:

```
from tensorflow_serving.apis import predict_pb2
...
request = predict_pb2.PredictRequest()
```

with:

```
from tensorflow_serving.apis import classification_pb2
...
request = classification_pb2.ClassificationRequest()
```

If you want to get predictions from a regression model, you can use the following imports:

```
from tensorflow_serving.apis import regression_pb2
...
regression_pb2.RegressionRequest()
```

Payloads

gRPC API uses Protocol Buffers as the data structure for the API request. By using binary Protocol Buffer payloads, the API requests use less bandwidth compared to JSON payloads. Also, depending on the model input data structure, you might experience faster predictions as with REST endpoints. The performance difference is explained by the fact that the submitted JSON data will be converted to a `tf.Example` data structure. This conversion can slow down the model server inference, and you might encounter a slower inference performance than in the gRPC API case.

Your data submitted to the gRPC endpoints needs to be converted to the protocol buffer data structure. TensorFlow provides you a handy utility function to perform the conversion called `tf.make_tensor_proto`. It allows various data formats, including scalars, lists, NumPy scalars, and NumPy arrays. The function will then convert the given Python or NumPy data structures to the protocol buffer format for the inference.

Model A/B Testing with TensorFlow Serving

A/B testing is an excellent methodology to test different models in real-life situations. In this scenario, a certain percentage of clients will receive predictions from model version A and all other requests will be served by model version B.

We discussed earlier that you could configure TensorFlow Serving to load multiple model versions and then specify the model version in your REST request URL or gRPC specifications.

TensorFlow Serving doesn't support server-side A/B testing, meaning that the model server will direct all client requests to a single endpoint to two model versions. But with a little tweak to our request URL, we can provide the appropriate support for random A/B testing from the client side:[7]

```
from random import random ❶

def get_rest_url(model_name, host='localhost', port=8501,
                 verb='predict', version=None):
    url = "http://{}:{}/v1/models/{}/".format(host, port, model_name)
    if version:
        url += "versions/{}".format(version)
    url += ":{}".format(verb)
    return url

...

# Submit 10% of all requests from this client to version 1.
# 90% of the requests should go to the default models.
threshold = 0.1
version = 1 if random() < threshold else None ❷
url = get_rest_url(model_name='complaints_classification', version=version)
```

❶ The `random` library will help us pick a model.

❷ If `version = None`, TensorFlow Serving will infer with the default version.

As you can see, randomly changing the request URL for our model inference (in our REST API example), can provide you some basic A/B testing functionality. If you would like to extend these capabilities by performing the random routing of the model inference on the server side, we highly recommend routing tools like Istio for this purpose. Originally designed for web traffic, Istio can be used to route traffic to specific models. You can phase in models, perform A/B tests, or create policies for data routed to specific models.

When you perform A/B tests with your models, it is often useful to request information about the model from the model server. In the following section, we will explain how you can request the metadata information from TensorFlow Serving.

Requesting Model Metadata from the Model Server

At the beginning of the book, we laid out the model life cycle and explained how we want to automate the machine learning life cycle. A critical component of the continuous life cycle is generating accuracy or general performance feedback about your model versions. We will take a deep dive into how to generate these feedback loops in Chapter 13, but for now, imagine that your model classifies some data (e.g., the sentiment of the text), and then asks the user to rate the prediction. The information of whether a model predicted something correctly or incorrectly is precious for improving future model versions, but it is only useful if we know which model version has performed the prediction.

The metadata provided by the model server will contain the information to annotate your feedback loops.

REST Requests for Model Metadata

Requesting model metadata is straightforward with TensorFlow Serving. TensorFlow Serving provides you an endpoint for model metadata:

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}[/versions/{MODEL_VERSION}]/metadata
```

Similar to the REST API inference requests we discussed earlier, you have the option to specify the model version in the request URL, or if you don't specify it, the model server will provide the information about the default model.

As Example 8-7 shows, we can request model metadata with a single `GET` request.

Example 8-7. Example model metadata request with a python client

```
import requests

def metadata_rest_request(model_name, host="localhost",
                          port=8501, version=None):
    url = "http://{}:{}/v1/models/{}/".format(host, port, model_name)
    if version:
        url += "versions/{}".format(version)
    url += "/metadata"  ❶
    response = requests.get(url=url) ❷
    return response
```

❶ Append `/metadata` for model information.

❷ Perform a `GET` request.

The model server will return the model specifications as a `model_spec` dictionary and the model definitions as a `metadata` dictionary:

```
{
  "model_spec": {
    "name": "complaints_classification",
    "signature_name": "",
    "version": "1556583584"
  },
  "metadata": {
    "signature_def": {
      "signature_def": {
        "classification": {
          "inputs": {
            "inputs": {
              "dtype": "DT_STRING",
              "tensor_shape": {
                ...
```

gRPC Requests for Model Metadata

Requesting model metadata with gRPC is almost as easy as the REST API case. In the gRPC case, you file a `GetModelMetadataRequest`, add the model name to the specifications, and submit the request via the `GetModelMetadata` method of the `stub`:

```
from tensorflow_serving.apis import get_model_metadata_pb2

def get_model_version(model_name, stub):
    request = get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name
    request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    return response.model_spec

model_name = 'complaints_classification'
stub = create_grpc_stub('localhost')
get_model_version(model_name, stub)

name: "complaints_classification"
version {
  value: 1556583584
}
```

The gRPC response contains a `ModelSpec` object that contains the version number of the loaded model.

More interesting is the use case of obtaining the model signature information of the loaded models. With almost the same request functions, we can determine the model's metadata. The only difference is that we don't access the `model_spec` attribute of the response object, but the `metadata`. The information needs to be serialized to be human readable; therefore, we will use `SerializeToString` to convert the protocol buffer information:

```
from tensorflow_serving.apis import get_model_metadata_pb2

def get_model_meta(model_name, stub):
    request = get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name
    request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    return response.metadata['signature_def']

model_name = 'complaints_classification'
stub = create_grpc_stub('localhost')
meta = get_model_meta(model_name, stub)

print(meta.SerializeToString().decode("utf-8", 'ignore'))
# type.googleapis.com/tensorflow.serving.SignatureDefMap
# serving_default
# complaints_classification_input
#         input_1:0
#                2@
# complaints_classification_output(
# dense_1/Sigmoid:0
#                tensorflow/serving/predict
```

gRPC requests are more complex than REST requests; however, in applications with high performance requirements, they can provide faster prediction performances. Another way of increasing our model prediction performance is by batching our prediction requests.

Batching Inference Requests

Batching inference requests is one of the most powerful features of TensorFlow Serving. During model training, batching accelerates our training because we can parallelize the computation of our training samples. At the same time, we can also use the computation hardware efficiently if we match the memory requirements of our batches with the available memory of the GPU.

If you run TensorFlow Serving without the batching enabled, as shown in Figure 8-3, every client request is handled individually and in sequence. If you classify an image, for example, your first request will infer the model on your CPU or GPU before the second request, third request, and so on, will be classified. In this case, we under-utilize the available memory of the CPU or GPU.

As shown in Figure 8-4, multiple clients can request model predictions, and the model server batches the different client requests into one "batch" to compute. Each request inferred through this batching step might take a bit longer than a single request because of the timeout or the limit of the batch. However, similar to our training phase, we can compute the batch in parallel and return the results to all clients after the completion of the batch computation. This will utilize the hardware more efficiently than single sample requests.

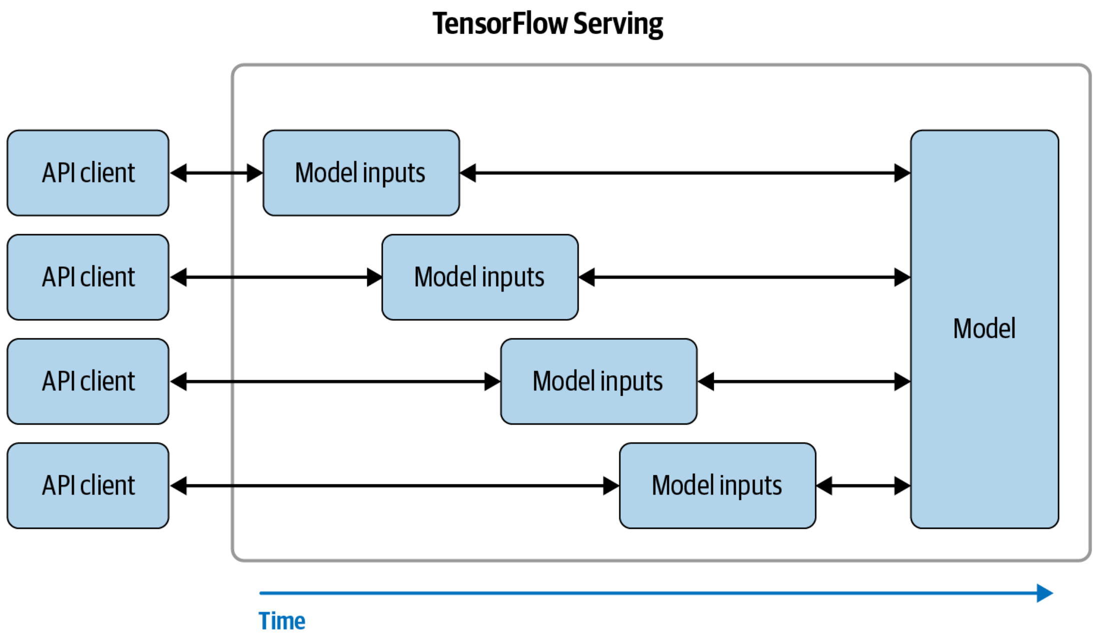


Figure 8-3. Overview of TensorFlow Serving without batching

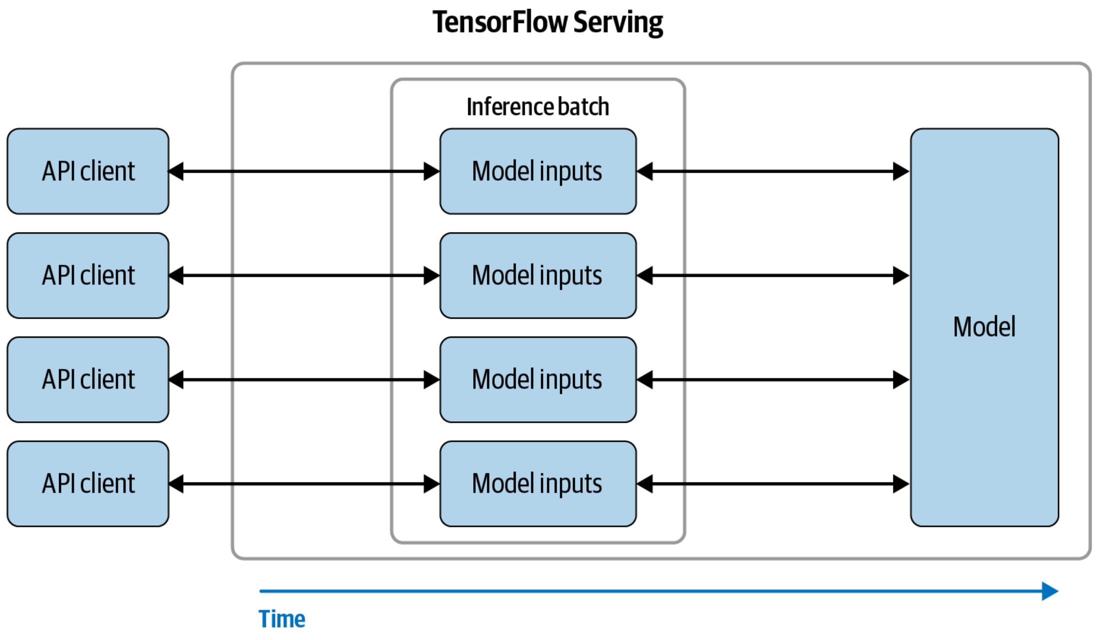


Figure 8-4. Overview of TensorFlow Serving with batching

Configuring Batch Predictions

Batching predictions needs to be enabled for TensorFlow Serving and then configured for your use case. You have five configuration options:

max_batch_size

This parameter controls the batch size. Large batch sizes will increase the request latency and can lead to exhausting the GPU memory. Small batch sizes lose the benefit of using optimal computation resources.

batch_timeout_micros

This parameter sets the maximum wait time for filling a batch. This parameter is handy to cap the latency for inference requests.

num_batch_threads

The number of threads configures how many CPU or GPU cores can be used in parallel.

max_enqueued_batches

This parameter sets the maximum number of batches queued for predictions. This configuration is beneficial to avoid an unreasonable backlog of requests. If the maximum number is reached, requests will be returned with an error instead of being queued.

pad_variable_length_inputs

This Boolean parameter determines if input tensors with variable lengths will be padded to the same lengths for all input tensors.

As you can imagine, setting parameters for optimal batching requires some tuning and is application dependent. If you run online inferences, you should aim for limiting the latency. It is often recommended to set `batch_timeout_micros` initially to 0 and tune the timeout toward 10,000 microseconds. In contrast, batch requests will benefit from longer timeouts (milliseconds to a second) to constantly use the batch size for optimal performance. TensorFlow Serving will make predictions on the batch when either the `max_batch_size` or the timeout is reached.

Set `num_batch_threads` to the number of CPU cores if you configure TensorFlow Serving for CPU-based predictions. If you configure a GPU setup, tune `max_batch_size` to get an optimal utilization of the GPU memory. While you tune your configuration, make sure that you set `max_enqueued_batches` to a huge number to avoid some requests being returned early without proper inference.

You can set the parameters in a text file, as shown in the following example. In our example, we create a configuration file called *batching_parameters.txt* and add the following content:

```
max_batch_size { value: 32 }
batch_timeout_micros { value: 5000 }
pad_variable_length_inputs: true
```

If you want to enable batching, you need to pass two additional parameters to the Docker container running TensorFlow Serving. To enable batching, set `enable_batching` to true and set `batching_parameters_file` to the absolute path of the batching configuration file inside of the container. Please keep in mind that you have to mount the additional folder with the configuration file if it isn't located in the same folder as the model versions.

Here is a complete example of the `docker run` command that starts the TensorFlow Serving Docker container with batching enabled. The parameters will then be passed to the TensorFlow Serving instance:

```
docker run -p 8500:8500 \
           -p 8501:8501 \
           --mount type=bind,source=/path/to/models,target=/models/my_model \
           --mount type=bind,source=/path/to/batch_config,target=/server_config
           -e MODEL_NAME=my_model -t tensorflow/serving \
           --enable_batching=true
           --batching_parameters_file=/server_config/batching_parameters.txt
```

As explained earlier, the configuration of the batching will require additional tuning, but the performance gains should make up for the initial setup. We highly recommend enabling this TensorFlow Serving feature. It is especially useful for inferring a large number of data samples with offline batch processes.

Other TensorFlow Serving Optimizations

TensorFlow Serving comes with a variety of additional optimization features. Additional feature flags are:

`--file_system_poll_wait_seconds=1`

TensorFlow Serving will poll if a new model version is available. You can disable the feature by setting it to `1`. If you only want to load the model once and never update it, you can set it to `0`. The parameter expects an integer value. If you load models from cloud storage buckets, we highly recommend that you increase the polling time to avoid unnecessary cloud provider charges for the frequent list operations on the cloud storage bucket.

`--tensorflow_session_parallelism=0`

TensorFlow Serving will automatically determine how many threads to use for a TensorFlow session. In case you want to set the number of a thread manually, you can overwrite it by setting this parameter to any positive integer value.

--tensorflow_intra_op_parallelism=0

This parameter sets the number of cores being used for running TensorFlow Serving. The number of available threads determines how many operations will be parallelized. If the value is `0`, all available cores will be used.

--tensorflow_inter_op_parallelism=0

This parameter sets the number of available threads in a pool to execute TensorFlow ops. This is useful for maximizing the execution of independent operations in a TensorFlow graph. If the value is set to `0`, all available cores will be used and one thread per core will be allocated.

Similar to our earlier examples, you can pass the configuration parameter to the `docker run` command, as shown in the following example:

```
docker run -p 8500:8500 \
           -p 8501:8501 \
           --mount type=bind,source=/path/to/models,target=/models/my_model \
           -e MODEL_NAME=my_model -t tensorflow/serving \
           --tensorflow_intra_op_parallelism=4 \
           --tensorflow_inter_op_parallelism=4 \
           --file_system_poll_wait_seconds=10 \
           --tensorflow_session_parallelism=2
```

The discussed configuration options can improve performance and avoid unnecessary cloud provider charges.

TensorFlow Serving Alternatives

TensorFlow Serving is a great way of deploying machine learning models. With the TensorFlow `Estimator`s and Keras models, you should be covered for a large variety of machine learning concepts. However, if you would like to deploy a legacy model or if your machine learning framework of choice isn't TensorFlow or Keras, here are a couple of options.

BentoML

BentoML is a framework-independent library that deploys machine learning models. It supports models trained through PyTorch, scikit-learn, TensorFlow, Keras, and XGBoost. For TensorFlow models, BentoML supports the `SavedModel` format. BentoML supports batching requests.

Seldon

The UK startup Seldon provides a variety of open source tools to manage model life cycles, and one of their core products is Seldon Core. Seldon Core provides you a toolbox to wrap your models in a Docker image, which is then deployed via Seldon in a Kubernetes cluster.

At the time of writing this chapter, Seldon supported machine learning models trained with TensorFlow, scikit-learn, XGBoost, and even R.

Seldon comes with its own ecosystem that allows building preprocessing into its own Docker images, which are deployed in conjunction with the deployment images. It also provides a routing service that allows you to perform A/B test or multiarm bandit experiments.

Seldon is highly integrated with the Kubeflow environment and, similar to TensorFlow Serving, is a way to deploy models with Kubeflow on Kubernetes.

GraphPipe

GraphPipe is another way of deploying TensorFlow and non-TensorFlow models. Oracle drives the open source project. It allows you to deploy not just TensorFlow (including Keras) models, but also Caffe2 models and all machine learning models that can be converted to the Open Neural Network Exchange (ONNX) format.[8] Through the ONNX format, you can deploy PyTorch models with GraphPipe.

Besides providing a model server for TensorFlow, PyTorch, etc., GraphPipe also provides client implementation for programming languages like Python, Java, and Go.

Simple TensorFlow Serving

Simple TensorFlow Serving is a development by Dihao Chen from 4Paradigm. The library supports more than just TensorFlow models. The current list of supported model frameworks includes ONNX, scikit-learn, XGBoost, PMML, and H2O. It supports multiple models, predictions on GPUs, and client code for a variety of languages.

One significant aspect of Simple TensorFlow Serving is that it supports authentication and encrypted connections to the model server. Authentication is currently not a feature of TensorFlow Serving, and supporting SSL or Transport Layer Security (TLS) requires a custom build of TensorFlow Serving.

MLflow

MLflow supports the deployment of machine learning models, but that it is only one aspect of the tool created by DataBricks. MLflow is designed to manage model experiments through MLflow Tracking. The tool has a built-in model server, which provides REST API endpoints for the models managed through MLflow.

MLflow also provides interfaces to directly deploy models from MLflow to Microsoft's Azure ML platform and AWS SageMaker.

Ray Serve

The Ray Project provides functionality to deploy machine learning models. *Ray Serve* is framework agnostic and it supports PyTorch, TensorFlow (incl. Keras), Scikit-Learn models, or custom model predictions. The library provides capabilities to batch requests and it allows the routing of traffic between models and their versions.

Ray Serve is integrated in the *Ray Project* ecosystem and supports distributed computation setups.

Deploying with Cloud Providers

All model server solutions we have discussed up to this point have to be installed and managed by you. However, all primary cloud providers—Google Cloud, AWS, and Microsoft Azure—offer machine learning products, including the hosting of machine learning models.

In this section, we will guide you through an example deployment using Google Cloud's AI Platform. Let's start with the model deployment, and later we'll explain how you can get predictions from the deployed model from your application client.

Use Cases

Managed cloud deployments of machine learning models are a good alternative to running your model server instances if you want to deploy a model seamlessly and don't want to worry about scaling the model deployment. All cloud providers offer deployment options with the ability to scale by the number of inference requests.

However, the flexibility of your model deployment comes at a cost. Managed services provide effortless deployments, but they cost a premium. For example, two model versions running full time (requiring two computation nodes) are more expensive than a comparable compute instance that is running a TensorFlow Serving instance. Another downside of managed deployments is the limitations of the products. Some cloud providers require that you deploy via their own software development kits, and others have limits on the node size and how much memory your model can take up. These limitations can be a severe restriction for sizeable machine learning models, especially if the models contain very many layers (i.e., language models).

Example Deployment with GCP

In this section, we will walk you through one deployment with Google Cloud's AI Platform. Instead of writing configuration files and executing terminal commands, we can set up model endpoints through a web UI.

LIMITS OF MODEL SIZE ON GCP'S AI PLATFORM

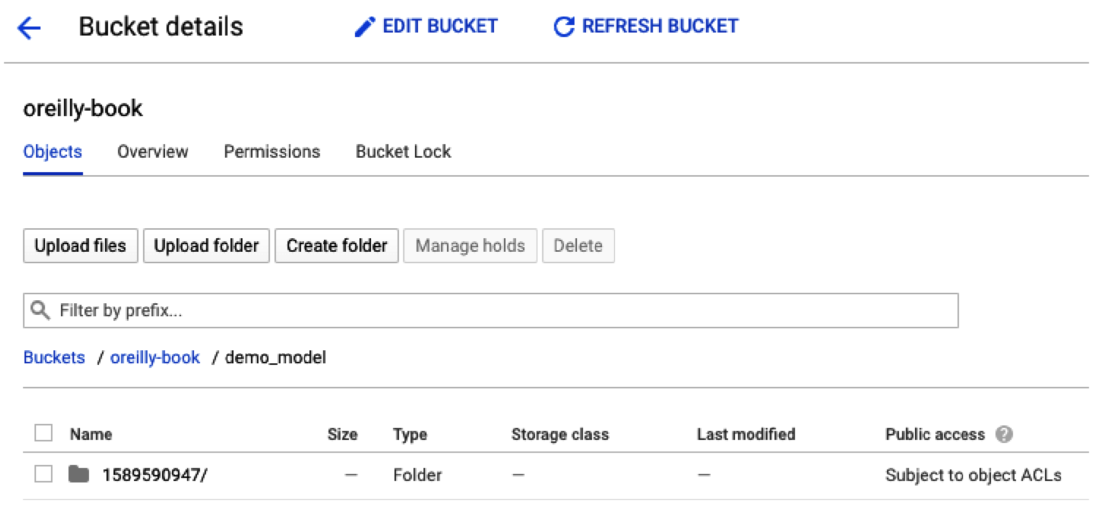
GCP's endpoints are limited to model sizes up to 500 MB. However, if you deploy your endpoints via compute engines of type *N1*, the maximum model limit is increased to 2 GB. At the time of writing, this option was available as a beta feature.

Model deployment

The deployment consists of three steps:

- Make the model accessible on Google Cloud.
- Create a new model instance with Google Cloud's AI Platform.
- Create a new version with the model instance.

The deployment starts with uploading your exported TensorFlow or Keras model to a storage bucket. As shown in Figure 8-5, you will need to upload the entire exported model. Once the upload of the model is done, please copy the complete path of the storage location.

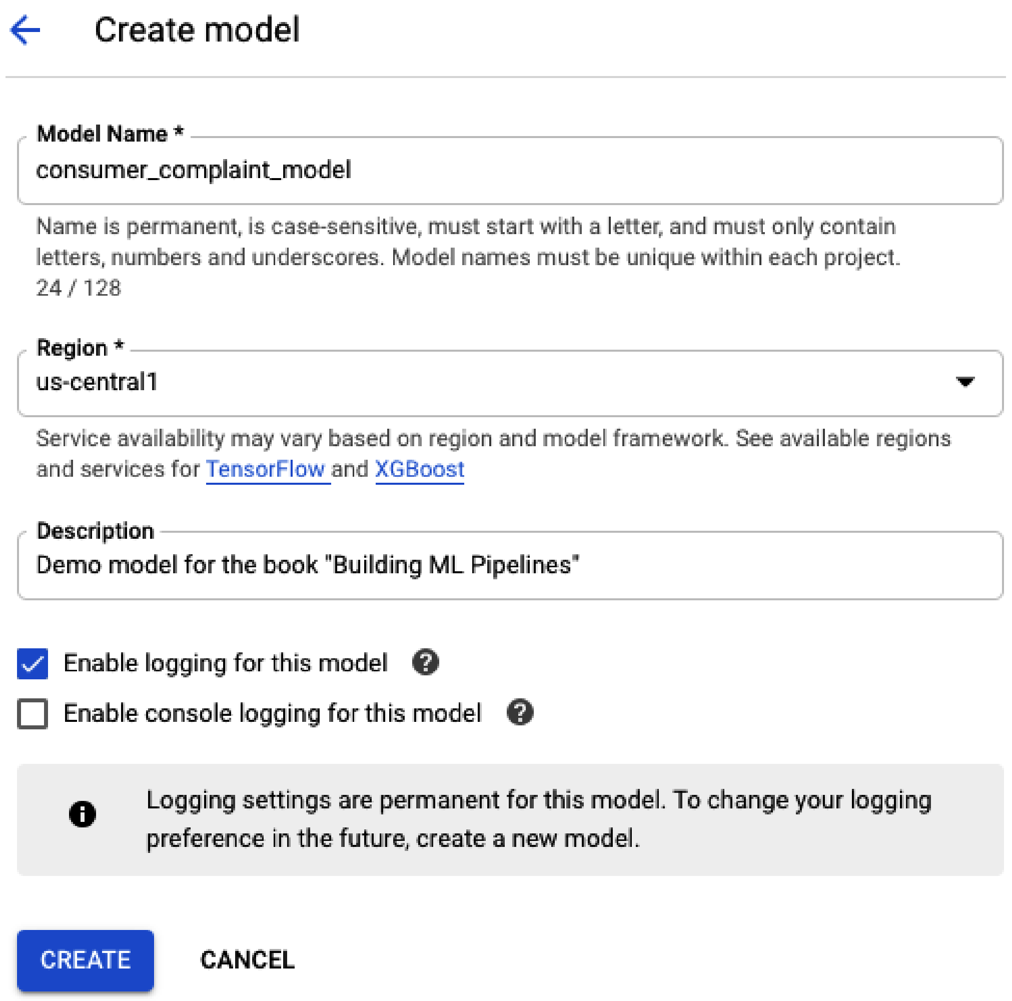


Name	Size	Type	Storage class	Last modified	Public access
1589590947/	–	Folder	–	–	Subject to object ACLs

Figure 8-5. Uploading the trained model to cloud storage

Once you have uploaded your machine learning model, head over to the AI Platform of GCP to set up your machine learning model for deployment. If it is the first time that you are using the AI Platform in your GCP project, you'll need to enable the API. The automated startup process by Google Cloud can take a few minutes.

As shown in Figure 8-6, you need to give the model a unique identifier. Once you have created the identifier, selected your preferred deployment region,[9] and created an optional project description, continue with the setup by clicking Create.


Create model

Model Name *
consumer_complaint_model
Name is permanent, is case-sensitive, must start with a letter, and must only contain letters, numbers and underscores. Model names must be unique within each project.
24 / 128

Region *
us-central1
Service availability may vary based on region and model framework. See available regions and services for TensorFlow and XGBoost

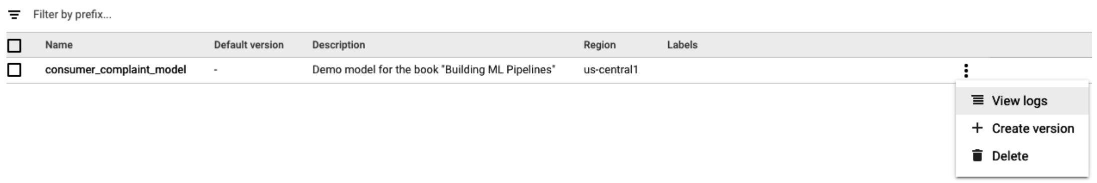
Description
Demo model for the book "Building ML Pipelines"

[✓] Enable logging for this model
[ ] Enable console logging for this model

Logging settings are permanent for this model. To change your logging preference in the future, create a new model.

Figure 8-6. Creating a new model instance

Once the new model is registered, the model will be listed in the dashboard, as shown in Figure 8-7. You can create a new model version for the dashboard by clicking "Create version" in the overflow menu.

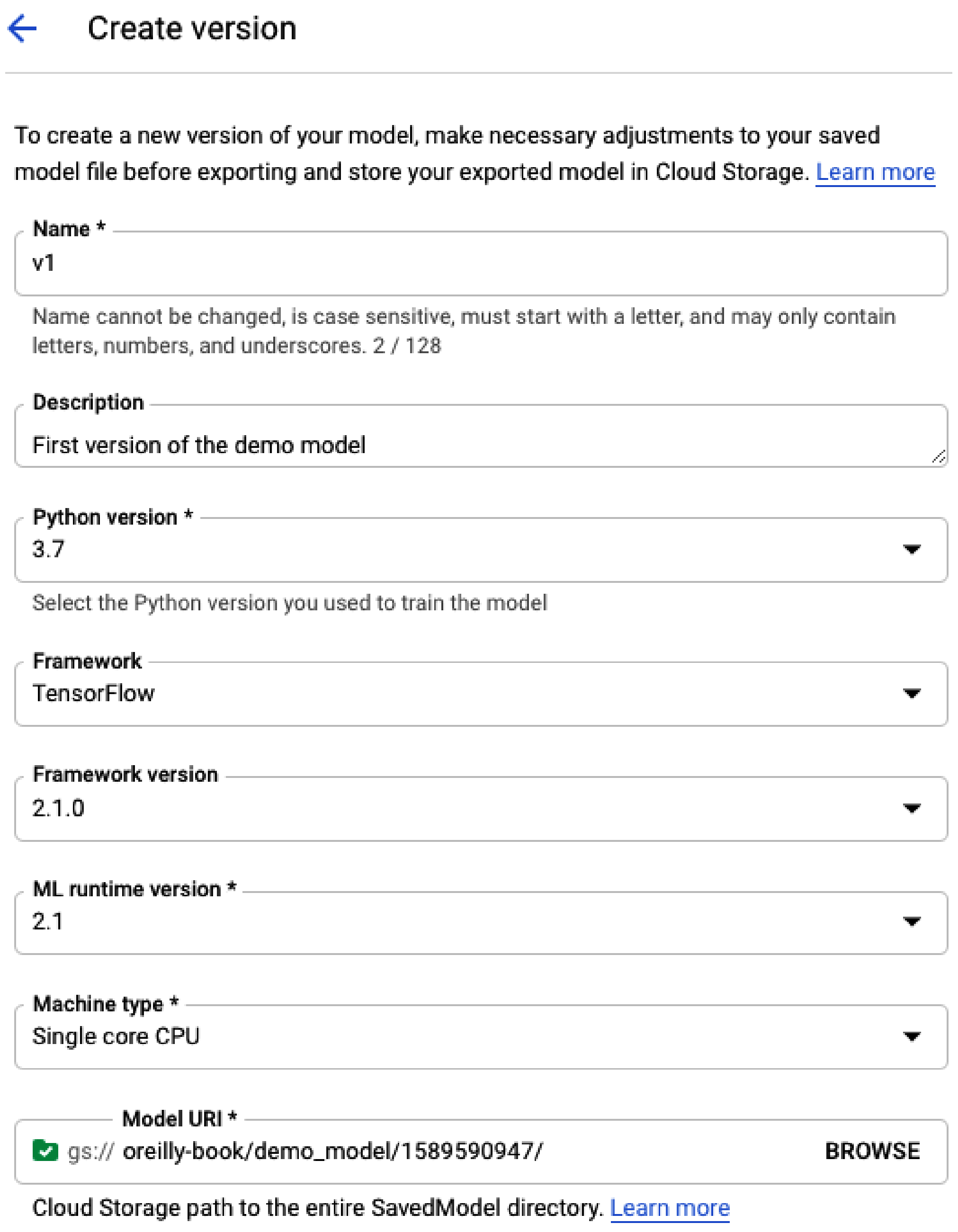

Filter by prefix...

Name	Default version	Description	Region	Labels
consumer_complaint_model	-	Demo model for the book "Building ML Pipelines"	us-central1	

Figure 8-7. Creating a new model version

When you create a new model version, you configure a compute instance that runs your model. Google Cloud gives you a variety of configuration options, as shown in Figure 8-8. The `version name` is important since you'll reference the `version name` later in the client setup. Please set the `Model URI` to the storage path you saved in the earlier step.

Google Cloud AI Platform supports a variety of machine learning frameworks including XGBoost, scikit-learn, and custom prediction routines.


Create version

To create a new version of your model, make necessary adjustments to your saved model file before exporting and store your exported model in Cloud Storage. Learn more

Name *: v1

Name cannot be changed, is case sensitive, must start with a letter, and may only contain letters, numbers, and underscores. 2 / 128

Description: First version of the demo model

Python version *: 3.7

Select the Python version you used to train the model

Framework: TensorFlow

Framework version: 2.1.0

ML runtime version *: 2.1

Machine type *: Single core CPU

Model URI *: gs:// oreilly-book/demo_model/1589590947/

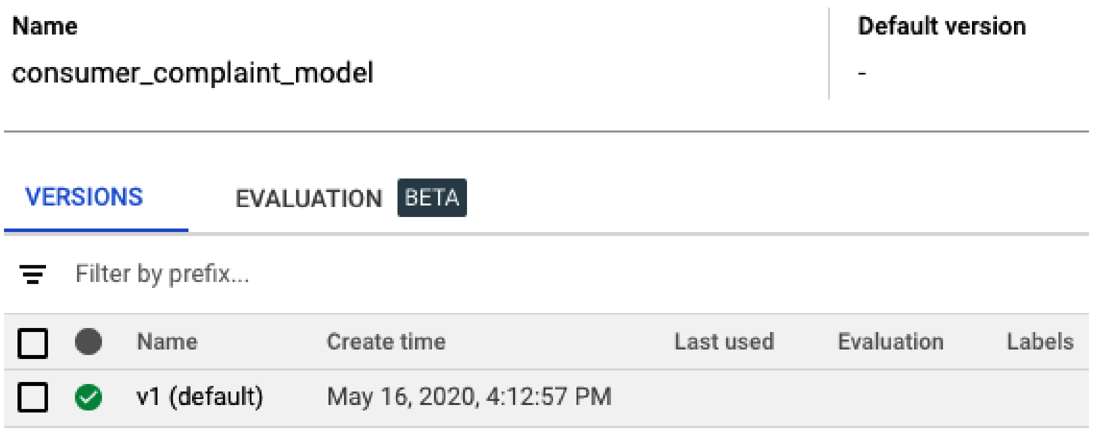
Cloud Storage path to the entire SavedModel directory. Learn more

Figure 8-8. Setting up the instance details

GCP also lets you configure how your model instance scales if your model experiences a large number of inference requests. You can select between two scaling behaviors: *manual scaling* or *autoscaling*.

Manual scaling gives you the option for setting the exact number of nodes available for the predictions of your model version. In contrast, autoscaling gives you the functionality to adjust the number of instances depending on the demand for your endpoint. If your nodes don't experience any requests, the number of nodes could even drop to zero. Please note that if autoscaling drops the number of nodes to zero, it will take some time to reinstantiate your model version with the next request hitting the model version endpoint. Also, if you run inference nodes in the autoscaling mode, you'll be billed in 10 min intervals.

Once the entire model version is configured, Google Cloud spins up the instances for you. If everything is ready for model predictions, you will see a green check icon next to the version name, as shown in Figure 8-9.



Name: consumer_complaint_model

Default version: -

		Name	Create time	Last used	Evaluation	Labels
☐	✓	v1 (default)	May 16, 2020, 4:12:57 PM			

Figure 8-9. Completing the deployment with a new version available

You can run multiple model versions simultaneously. In the control panel of the model version, you can set one version as the default version, and any inference request without a version specified will be routed to the designated "default version." Just note that each model version will be hosted on an individual node and will accumulate GCP costs.

Model inference

Since TensorFlow Serving has been battle tested at Google and is used heavily internally, it is also used behind the scenes at GCP. You'll notice that the AI Platform isn't just using the same model export format as we have seen with our TensorFlow Serving instances but the payloads have the same data structure as we have seen before.

The only significant difference is the API connection. As you'll see in this section, you'll connect to the model version via the GCP API that is handling the request authentication.

To connect with the Google Cloud API, you'll need to install the library `google-api-python-client` with:

```
$ pip install google-api-python-client
```

All Google services can be connected via a service object. The helper function in the following code snippet highlights how to create the service object. The Google API client takes a `service name` and a `service version` and returns an object that provides all API functionalities via methods from the returned object:

```
import googleapiclient.discovery

def _connect_service():
    return googleapiclient.discovery.build(
        serviceName="ml", version="v1"
    )
```

Similar to our earlier REST and gRPC examples, we nest our inference data under a fixed `instances` key, which carries a list of input dictionaries. We have created a little helper function to generate the payloads. This function contains any preprocessing if you need to modify your input data before the inference:

```
def _generate_payload(sentence):
    return {"instances": [{"sentence": sentence}]}
```

With the service object created on the client side and the payload generated, it's time to request the prediction from the Google Cloud–hosted machine learning model.

The service object of the AI Platform service contains a predict method that accepts a `name` and a `body`. The name is a path string containing your GCP project name, model name, and, if you want to make predictions with a specific model version, version name. If you don't specify a version number, the default model version will be used for the model inference. The body contains the inference data structure we generated earlier:

```
project = "yourGCPProjectName"
model_name = "demo_model"
```

```
version_name = "v1"
request = service.projects().predict(
    name="projects/{}/models/{}/versions/{}".format(
        project, model_name, version_name),
    body=_generate_payload(sentence)
)
response = request.execute()
```

The Google Cloud AI Platform response contains the predict scores for the different categories similar to a REST response from a TensorFlow Serving instance:

```
{'predictions': [
    {'label': [
        0.9000182151794434,
        0.02840868942439556,
        0.009750653058290482,
        0.06182243302464485
    ]}
]}
```

The demonstrated deployment option is a quick way of deploying a machine learning model without setting up an entire deployment infrastructure. Other cloud providers like AWS or Microsoft Azure offer similar model deployment services. Depending on your deployment requirements, cloud providers can be a good alternative to self-hosted deployment options. The downsides are potentially higher costs and the lack of completely optimizing the endpoints (i.e., by providing gRPC endpoints or batching functionality, as we discussed in "Batching Inference Requests").

Model Deployment with TFX Pipelines

In the introduction to this chapter, in Figure 8-1 we showed the deployment steps as one component of a machine learning pipeline. After discussing the inner workings of model deployments, and especially TensorFlow Serving, we want to connect the dots with our machine learning pipeline in this section.

In Figure 8-10, you can see the steps for a continuous model deployment. We assume that you have TensorFlow Serving running and configured to load models from a given file location. Furthermore, we assume that TensorFlow Serving will load the models from an external file location (i.e., a cloud storage bucket or a mounted persistent volume). Both systems, the TFX pipeline and the TensorFlow Serving instance, need to have access to the same filesystem.

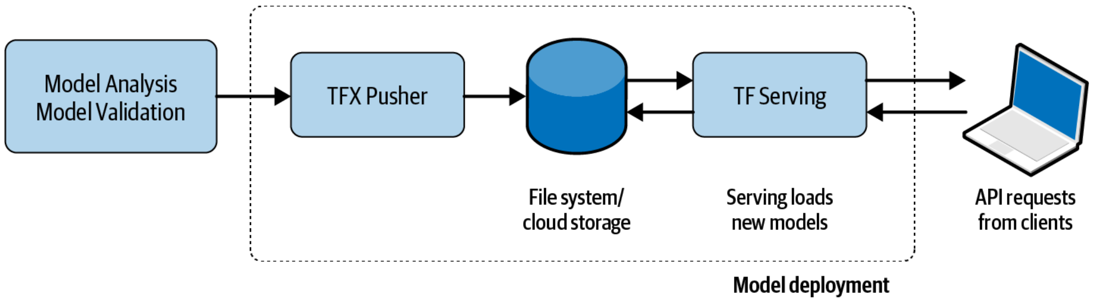


Figure 8-10. Deployment of models produced from TFX pipelines

In "TFX Pusher Component", we discussed the `Pusher` component. The TFX component allows us to push validated models to a given location (e.g., a cloud storage bucket). TensorFlow Serving can pick up new model versions from the cloud storage location, unload the earlier model version, and load the latest version for the given model endpoint. This is the default model policy of TensorFlow Serving.

Due to the default model policy, we can build a simple continuous deployment setup with TFX and TensorFlow Serving fairly easily.

Summary

In this chapter, we discussed how to set up TensorFlow Serving to deploy machine learning models and why a model server is a more scalable option than deploying machine learning models through a Flask web application. We stepped through the installation and configuration steps, introduced the two main communication options, REST and gRPC, and briefly discussed the advantages and disadvantages of both communication protocols.

Furthermore, we explained some of the great benefits of TensorFlow Serving, including the batching of model requests and the ability to obtain metadata about the different model versions. We also discussed how to set up a quick A/B test setup with TensorFlow Serving.

We closed this chapter with a brief introduction of a managed cloud service, using Google Cloud AI Platform as an example. Managed cloud services provide you the ability to deploy machine learning models without managing your own server instances.

In the next chapter, we will discuss enhancing our model deployments, for example, by loading models from cloud providers or by deploying TensorFlow Serving with Kubernetes.

1 For application use cases, visit TensorFlow.

2 Model *tag sets* are used to identify MetaGraphs for loading. A model could be exported with a graph specified for training and serving. Both MetaGraphs can be provided through different model tags.

3 If you haven't installed or used `Docker` before, check out our brief introduction in Appendix A.

4 For a more detailed introduction to REST and gRPC, please check "REST Versus gRPC".

5 The loading and unloading of models only works if the `file_system_poll_wait_seconds` is configured to be greater than 0. The default configuration is 2s.

6 The SSL configuration file is based on the SSL configuration protocol buffer, which can be found in the TensorFlow Serving API.

7 A/B testing isn't complete without the statistical test of the results you get from people interacting with the two models. The shown implementation just provides us the A/B testing backend.

8 ONNX is a way of describing machine learning models.

9 For the lowest prediction latencies, choose a region closest to the geographical region of the model requests.