available and how they can be run in ways to ensure scalability, reliability, and availability.

Chapter 14. Model Serving Examples

This chapter provides three examples that take a hands-on approach to serving ML models effectively and efficiently. In the first example, we'll take a deep dive into the deployment of TensorFlow and JAX models. In the second example, we'll address how you can optimize your deployment setup with TensorFlow Profiler.

For our third example, we will introduce TorchServe, the model deployment setup for Torch-based models.

Example: Deploying TensorFlow Models with TensorFlow Serving

Using machine framework–specific deployment libraries through Python API implementations provides a number of performance benefits. In this example, we'll focus on TensorFlow Serving (TF Serving), which allows you to deploy TensorFlow, Keras, JAX, and scikit-learn models effectively. If you're interested in how to deploy PyTorch models, hop over to this chapter's third example, where we'll be focusing on TorchServe, the PyTorch-specific deployment library.

Let's assume you have trained, evaluated, and exported a TensorFlow/Keras model. In this section, we'll introduce how you can set up a TF Serving

instance with Docker, show how to configure TF Serving, and then demonstrate how you can request predictions from the model server.

Exporting Keras Models for TF Serving

Before deploying your ML model, you need to export it. TF Serving supports the TensorFlow SavedModel format, which is serializing the model into a protocol buffer format. The following example shows how to export a TensorFlow or Keras model to the SavedModel format:

```
import tensorflow as tf

my_model = ...
# Convert the Keras model to a TF SavedModel
tf.keras.models.save_model(my_model, '/tmp/models
```

We can now consume the exported model in TF Serving.

Setting Up TF Serving with Docker

The easiest way to run TF Serving is through prebuilt Docker images. If your model can run on CPUs, you can use the following `docker` command:

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

If your model requires GPU support, use the following `docker` command to load the latest image containing the matching CUDA drivers:

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

You can also install TF Serving natively on Linux operating systems. For detailed installation instructions, refer to the TensorFlow Serving documentation.

For now, let's focus on the basic TF Serving configuration.

Basic Configuration of TF Serving

The basic configuration of TF Serving is straightforward. TF Serving needs a base path to know where to look for ML models, and the name of the model to load. TF Serving will then detect the latest model version (based on the subfolder name) and load the most recent model. Therefore, it is advised to export models with the epoch timestamp of the export time as the folder name. That's all you need to know for the basic TF Serving configuration.

All other configuration details in our example set the Docker configuration for us to access TF Serving. The first two configuration parameters set the

shared ports between the host machine and the Docker container. The third configuration sets the mount path so that the container can access a folder on the host machine. That simplifies model loading, because otherwise, the model could have to be “backed” into the Docker image during build time:

```
$ docker run -p 8500:8500 \ ❶
             -p 8501:8501 \
             --mount type=bind,source=/tmp/models
             -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.

For local deployment/testing, local ports are mapped to container ports, and the model directory from localhost is mounted into the container with a model name passed via environment variables.

Once your serving container is starting up, you should see output on your terminal that is similar to the following:

```
2023-07-26 07:26:20: I tensorflow_serving/model_s
  Building single TensorFlow model file config:
  model_name: my_model model_base_path: /models/m
2023-07-26 07:26:20: I tensorflow_serving/model_s
  Adding/updating models.
2023-07-26 07:26:20: I tensorflow_serving/model_s
  (Re-)adding model: my_model
...
2023-07-26 07:26:34: I tensorflow_serving/core/lo
  Successfully loaded servable version {name: my_
2023-07-26 07:26:34: I tensorflow_serving/model_s
  Running gRPC ModelServer at 0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename n
[evhttp_server.cc : 237] RAW: Entering the event
2023-07-26 07:26:34: I tensorflow_serving/model_s
  Exporting HTTP/REST API at:localhost:8501 ...
```

TF Serving allows a number of additional configuration options.

Making Model Prediction Requests with 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 *requests* library to handle the HTTP requests:

```
$ pip install requests
```

The following example showcases an example POST request:

```
import requests

def get_rest_request(text, model_name="my_model"
    url = "http://localhost:8501/v1/models/{}:pre
    payload = {"instances": [text]} ❷
    response = requests.post(url=url, json=payloa
    return response
rs_rest = get_rest_request(text="classify my text
rs_rest.json()
```

❶ Replace `localhost` with an IP address if the server isn't running on the same machine.

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

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}`. Here is a summary of that information:

`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}`.

Making Model Prediction Requests with 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 that replicates the available methods from the server:

```
import grpc
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_se
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.Predictio
    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='
                 signature_name='classification'
    request = predict_pb2.PredictRequest()
    request.model_spec.name = model_name
    request.model_spec.signature_name = signature
    request.inputs['inputs'].CopyFrom(tf.make_te

    result_future = stub.Predict.future(request,
    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 functions now available, we can infer our example datasets with these 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_c
credentials = grpc.ssl_channel_credentials(
    ca_cert, key, cert
)
channel = implementations.secure_channel(hostport
```

On the server side, TF Serving can terminate secure connections if the Secure Sockets Layer (SSL) protocol is configured. To terminate secure connections, create an SSL configuration file as shown in the following example:

```
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 filepath to the TF Serving argument `--ssl_config_file` during the start of TF Serving:

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --model_name=my_model \
                          --model_base_path=/mode
                          --ssl_config_file="<pa
```

Getting Predictions from Classification and Regression Models

If you're 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 these:

```
from tensorflow_serving.apis import classificatio
...
request = classification_pb2.ClassificationReques
```

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

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

Using Payloads

The 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 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.

Getting Model Metadata from TF Serving

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

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}[/vers
```

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.

We can request the model metadata with a single GET request:

```
import requests
def metadata_rest_request(model_name, host="loca
                          port=8501, version=None
    url = "http://{}:{}/v1/models/{}/".format(hos
    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": "text_classification",
    "signature_name": "",
    "version": "1556583584"
  },
  "metadata": {
```

```
    "signature_def": {
      "signature_def": {
        "classification": {
          "inputs": {
            "inputs": {
              "dtype": "DT_STRING",
              "tensor_shape": {
                ...
```

Making Batch Inference Requests

Batching predictions needs to be enabled for TF 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 whether 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 try to limit the latency. For example, 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. TF 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 TF 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 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 TF Serving. To enable batching, set `enable_batching` to `true` and set `batching_parameters_file` to the absolute path of the batching configuration file inside the container. 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 TF Serving Docker container with batching enabled. The parameters will then be passed to the TF Serving instance:

```
docker run -p 8500:8500 \
           -p 8501:8501 \
           --mount type=bind,source=/path/to/mode
           --mount type=bind,source=/path/to/batc
           -e MODEL_NAME=my_model -t tensorflow/s
           --enable_batching=true
           --batching_parameters_file=/server_con
```

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

Example: Profiling TF Serving Inferences with TF Profiler

With the growing complexity of today's deep learning models, the aspect of model inference latency is more relevant than ever. Therefore, profiling your ML model for bottlenecks can save you milliseconds during your prediction requests, and it will ultimately save you real money when it comes to deploying your model in a production scenario (and CO_2 emissions too).

TensorFlow and Keras models can be profiled with TensorBoard, which provides a number of tools to let you take a deep dive into your ML model. Keras already provides a stellar callback function to hook the training up to TensorBoard. This connection allows you to profile your model's performance during the training phase. However, this profiler setup tells you only half the story.

If you use the TensorBoard callback to profile your ML model, all TensorFlow ops used during the backward pass will be part of the profiling statistics. For example, you'll find optimizer ops muddled in those profiling statistics, and some of the ops might show a very different profile because they are executed on a GPU instead of a CPU. The information is extremely helpful if you want to optimize for more efficient training patterns, but it is less helpful in reducing your serving latency.

One of the many amazing features of TF Serving is the integrated TensorFlow Profiler. TF Profiler can connect to your TF Serving instance and profile your inference requests. Through this setup, you can investigate all inference-related ops and it will mimic the deployment scenario better than profiling your model during the training phase.

Prerequisites

For the purpose of this example, let's create a demo model based on the following code. Don't replicate the model, but rather make sure you export

your TensorFlow or JAX model in the SavedModel format that TF Serving can load:

```
import tensorflow as tf
import tensorflow_text as _
import tensorflow_hub as hub

text_input = tf.keras.layers.Input(shape=(), dtyp
preprocessor = hub.KerasLayer(
    "https://tfhub.dev/tensorflow/bert_en_uncased
encoder_inputs = preprocessor(text_input)
encoder = hub.KerasLayer(
    "https://tfhub.dev/tensorflow/bert_en_uncased
    trainable=True)
outputs = encoder(encoder_inputs)
sequence_output = outputs["sequence_output"]
embedding_model = tf.keras.Model(text_input, sequ
embedding_model.save("/models/test_model/1/")
```

TensorBoard Setup

Once you have your model saved in a location where TF Serving can load it from, you need to set up TS Serving and TensorBoard. First, let’s create a Docker image to host TensorBoard.

TensorBoard doesn't ship with the profiler anymore, so you need to install it separately. Once you create the Docker image, you can use `docker compose` to spin up TF Serving together with the newly created TensorBoard image:

```
FROM tensorflow/tensorflow:${TENSORFLOW_SERVING_V
RUN pip install -U tensorboard-plugin-profile
ENTRYPOINT [\"/usr/bin/python3\", \"-m\", \"tenso
\"/tmp/tensorboard\", \"--bind_all\"]
```

Our *docker-compose.yml* file looks like this:

```
version: '3.3'
services:
  ${TF_SERVING_HOSTNAME}:
    image: tensorflow/serving:${TF_SERVING_VERSIO
    ports:
      - '8500:8500'
      - '8501:8501'
    environment:
      - MODEL_NAME=${TF_SERVING_MODEL_NAME}
    hostname: '${TF_SERVING_HOSTNAME}'
    volumes:
      - '/models/${TF_SERVING_MODEL_NAME}:/models
      - '${TENSORBOARD_LOGDIR}:/tmp/tensorboard'
    command:
```

```
        - '--xla_cpu_compilation_enabled'
        - '--tensorflow_intra_op_parallelism=${INTR
        - '--tensorflow_inter_op_parallelism=${INTR
    profiler:
      image: ${DOCKER_PROFILER_TAG}
      ports:
        - '6006:6006'
      volumes:
        - '${TENSORBOARD_LOGDIR}:/tmp/tensorboard'
```

It’s useful to add TF Serving commands to the Docker configuration to mimic the full production setup as closely as possible. In this particular case, we enabled XLA support and limited the intra- and interops parallelism in TF Serving (you can find more information about XLA in the XLA developer guide and details about all TF Serving options on the TensorFlow website):

```
  command:
      - '--xla_cpu_compilation_enabled'
      - '--tensorflow_intra_op_parallelism=${INTRA_
      - '--tensorflow_inter_op_parallelism=${INTER_
```

Model Profile

If you execute the Docker containers, it will start up a TF Serving instance that loads your model (adjusts the model path in the script) and a TensorBoard instance as well.

If you're running this script remotely, you need to create an SSH tunnel to access TensorBoard. If you're running on a Google Cloud instance, you can do this by running the following command:

```
$ gcloud compute ssh \
    --project=digits-data-science \
    --zone=us-central1-a \
    YOUR_INSTANCE_NAME
```

More information about connecting securely to Google Cloud instances can be found in the Google Cloud documentation.

If you run the `docker compose` setup on your machine locally, you can skip the previous step. If you are running on an AWS EC2 instance, check the AWS documentation on how to connect with your machine.

Once `docker compose` is running, you should see a terminal output similar to the following. If the serving or profiler container fails with an error, you'll need to stop here and investigate. Both containers are needed for the next steps:

```
$ sh ./tensorboard.sh
mkdir -p /tmp/tensorboard
[+] Building 0.0s (6/6) FINISHED
=> [internal] load build definition from Dockerf
=> transferring dockerfile:
=> [internal] load dockerignore
=> [internal] load metadata for docker.io/tensor
=> [1/2] FROM docker.io/tensorflow/tensorflow:2.1
 => CACHED [2/2] RUN pip install -U tensorboard-p
=> exporting to image
=> => exporting layers
...
=> => naming to docker.io/library/tensorboard_pro
Starting 20230128_tfserving_profiling_serving_1
Recreating 20230128_tfserving_profiling_profiler_
Attaching to 20230128_tfserving_profiling_serving
serving_1   | 2023-02-12 18:30:46.059050: I
... Building single TensorFlow model file config
model_name: test_model model_base_path: /models/t
... serving_1   | 2023-02-12 18:30:48.495900: I
... Running initialization op on SavedModel bundl
/models/test_model/1 serving_1   | 2023-02-12 18
I ... SavedModel load for tags { serve }; Status
Took 2803691 microseconds. ...
234 | Chapter 14: Model-Serving Examplesserving_1
2023-02-12 18:30:49.296815: I ... Profiler servic
| 2023-02-12 18:30:49.298806: I ... Running gRPC
0.0.0.0:8500 ...
```

```
  serving_1    | [warn] getaddrinfo: address family
  serving_1    | 2023-02-12 18:30:49.300120: I ...
  Exporting HTTP/REST API at:localhost:8501 ...
  serving_1    | [evhttp_server.cc : 245] NET_LOG: E
```

If both containers are running, go to your browser and access *http://localhost:6006*. You can start the TensorBoard Profiler by selecting PROFILE from the top-right menu, as shown in Figure 14-1.

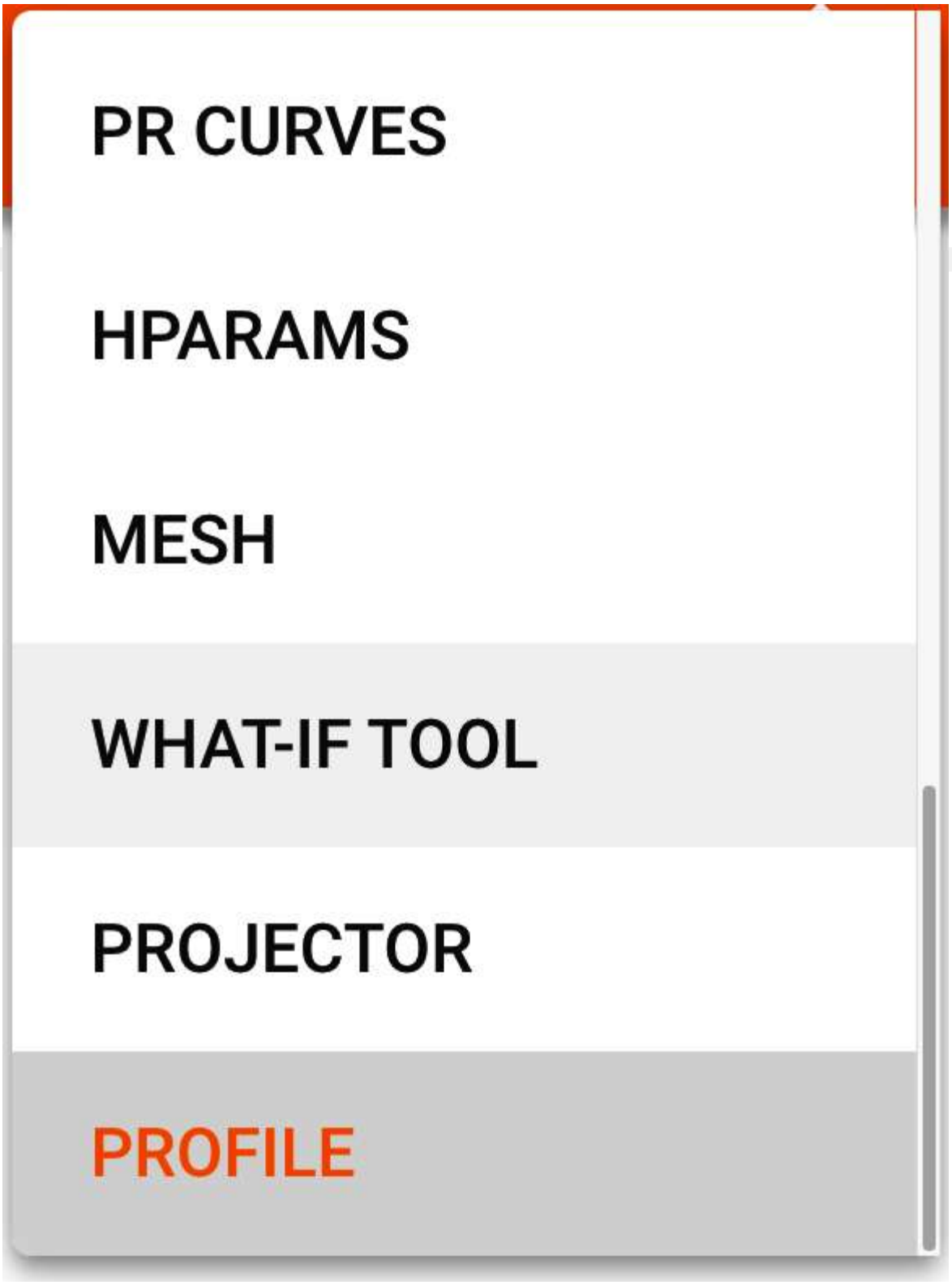

Figure 14-1. TensorFlow Profiler menu options

If you select PROFILE, it will open a menu to configure your Profiler session, shown in Figure 14-2. If you use the provided script, the hostname is `serving`. By default, TensorBoard profiles for 1 second. This is fairly short, and it takes some time to kick off an inference; 4,000 ms as a profiling duration is recommended.

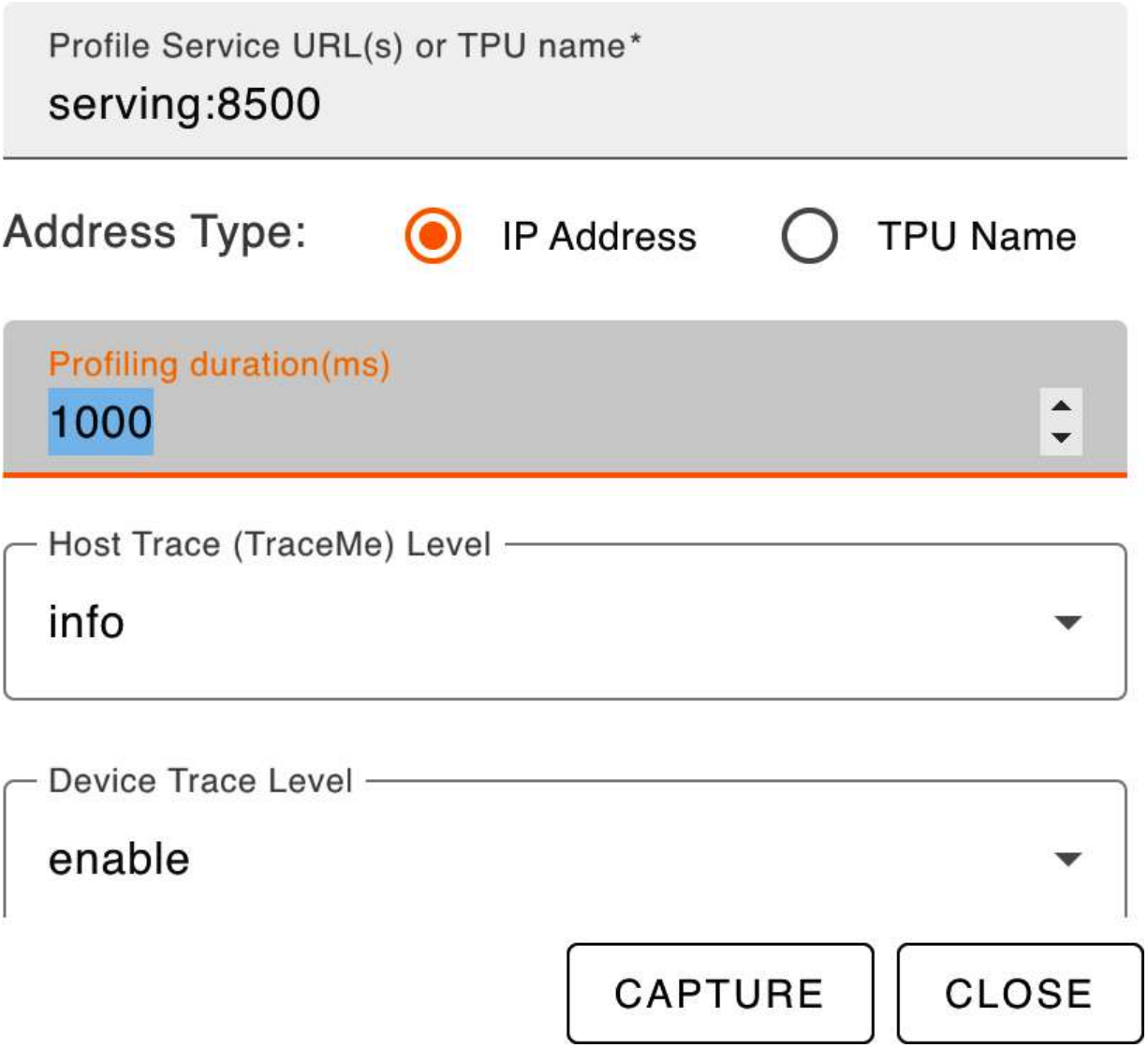

Figure 14-2. TensorFlow Profiler settings

Select CAPTURE, and then submit a prediction request to your TF Serving setup. You can do this with the following `curl` command:

```
$ curl -X POST \
       --data @data.json \
       http://localhost:8501/v1/models/test
```

If your payload is more than a few characters, save it in a JSON-formatted file (here, *data.json*). The `curl` command can load the file and submit it as the request payload:

```
$ curl -X POST \
       --data @data.json \
       http://localhost:8501/v1/models/test_model
```

A few seconds after you submit your curl request, you'll be provided with a variety of profiling details in TensorBoard. The TensorFlow Stats and the Tracer are the most insightful. The TensorFlow Stats tell you what ops are used most often. This provides you with details on how you could optimize your ML model. The Tracer shows every TensorFlow ops in its sequence. In Figure 14-3, you can see the trace of a BERT model with its 12 layers.

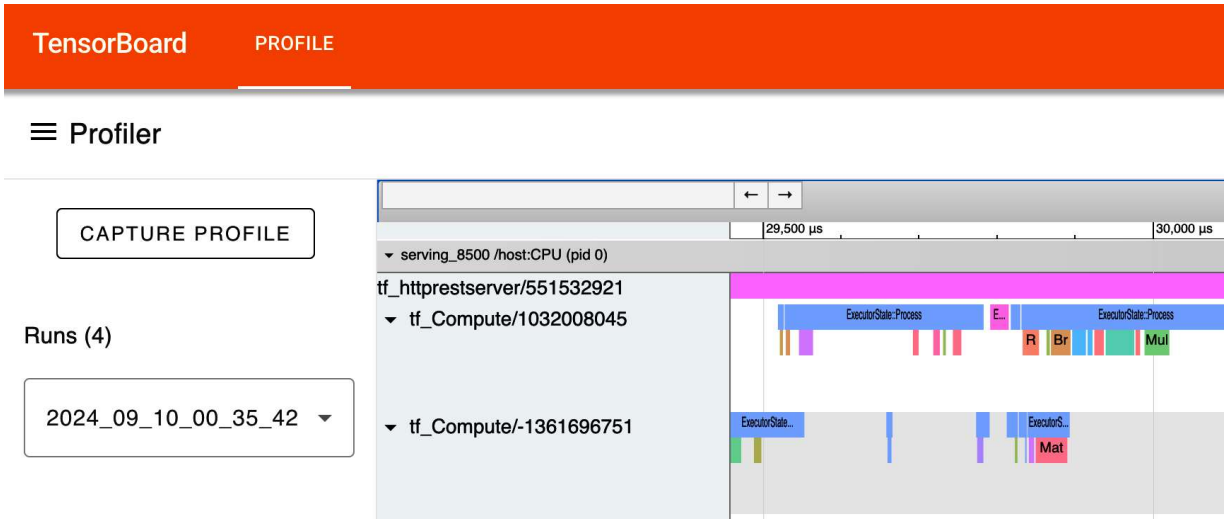

Figure 14-3. Screenshot of the TensorFlow Profiler results[1]

You can then zoom into any section of interest (see Figure 14-4). For example, we are always checking how much time is taken up by the preprocessing step in the model.

Figure 14-4. Zooming in to the results[2]

You can then click on every ops and drill into the specific details (see Figure 14-5). You might be surprised by what you discover.

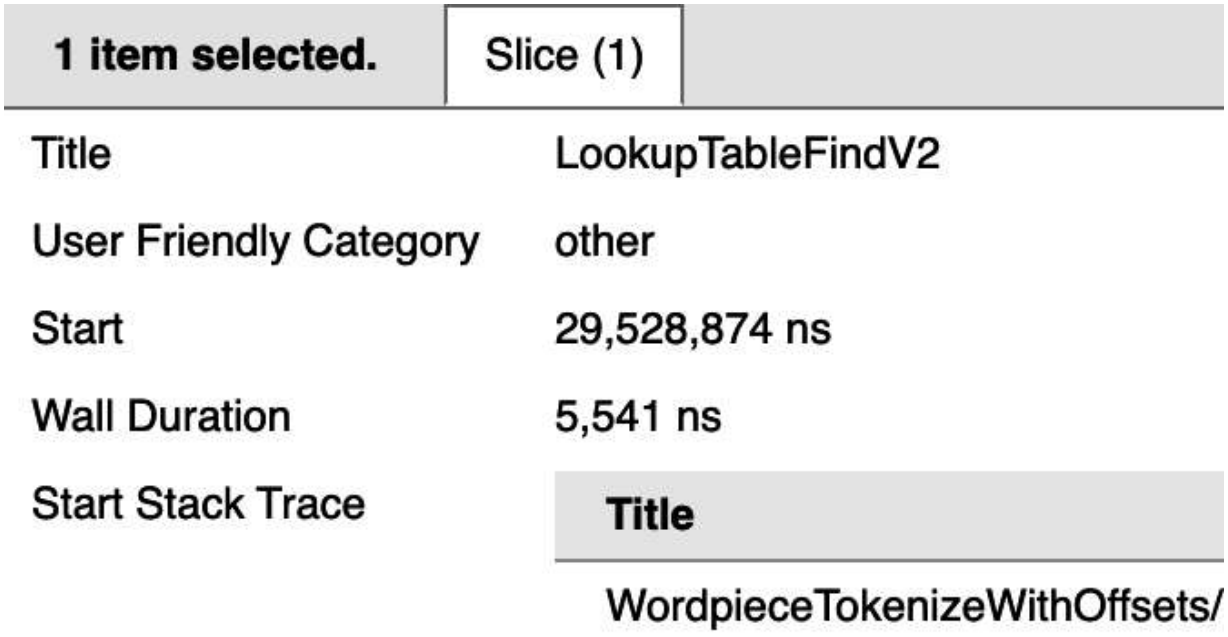

Figure 14-5. Ops details shown in the TensorFlow Profiler

Example: Basic TorchServe Setup

In this section, we will introduce the deployment of PyTorch models with TorchServe. While TF Serving supports a number of ML frameworks, it doesn’t support PyTorch model deployments. Fortunately, the PyTorch community has created a good alternative for PyTorch model deployments, called TorchServe, that follows the core principles of model deployments (e.g., consistent model requests, batch inferences). In the following sections, we’ll walk you through the necessary steps to deploy your PyTorch model. To simplify the experiment, we’ll deploy a generic PyTorch model.

Unlike TF Serving, TorchServe doesn't consume a model graph and is purely C++ based. TorchServe loads Python-based handlers and orchestrates the Python handler via a Java backend. But no worries, you won't have to write Java code to deploy your ML models.

Installing the TorchServe Dependencies

First, let's install the TorchServe dependencies. This will require Torch and all Python libraries to perform a model prediction. Then, we'll install the `torchserve` package and the `torch-model-archiver`:

```
$ pip install torch torchtext torchvision sentenc
$ pip install torchserve torch-model-archiver
```

Exporting Your Model for TorchServe

In the previous step, we installed the `torch-model-archiver`. The helper library creates a model archive by bundling all model files into a single compressed file. That way, the model can be easily deployed and no files are accidentally left behind.

First, serialize your PyTorch model with the following Python command:

```
> torch.save(model, '/my_model/model.py')
```

With the model now being serialized, you can create the model archive with the following shell command:

```
$ torch-model-archiver --model-name my_model \
                       --version 1.0 \
                       --model-file /my_model/mod
                       --serialized-file /my_mode
                       --extra-files /my_model/ir
                       --handler my_classifier
```

The archiver will create an archive file *.mar* that we can then deploy with TorchServe. Following are a few notes regarding the command arguments:

`model-name`

Defines the name of your model.

`version`

A version identifier defined by you.

`model-file`

Contains the Python code to load the ML model for the prediction (more in the next section).

`serialized-file`

The reference to the serialized model we created in the previous step.

`handler`

The name TorchServe will use to host the model under.

Setting Up TorchServe

The model archive can now be deployed with the following command:

```
$ torchserve --start \
             --model-store model_store \
             --models my_model=my_model.mar
```

The `model-store` points toward the location for all your archives, and `models` is the name of the model archive to be hosted. Once you start up TorchServe, you'll see output similar to the following:

```
$ torchserve OUTPUT
```

Request handlers

TorchServe performs model predictions through so-called *handlers*. These are Python classes with common prediction functionality such as initialization, preprocessing, prediction, and postprocessing functions.

TorchServe provides a number of basic handlers for text or image classifications. The following code example shows how you can write your own request handler:

```
import torch
import torch.nn as nn

from ts.torch_handler.base_handler import BaseHa
class MyModelHandler(BaseHandler):

    def __init__(self):
        self._context = None
        self.initialized = False
        self.model = None

    def initialize(self, context):
        self._context = context
        self.model = torch.load(context.system_pr
        self.model.eval()
        self.initialized = True

    def transforms(self, data):
        # your transformations go here
        …
        return data

    def preprocess(self, data):
        data = data[0].get("data") or data[0].ge
        tensor = self.transforms(data)
        return tensor.unsqueeze(0)

    def inference(self, data):
```

```
            with torch.no_grad():
                output = self.model(data)
                _, predicted = torch.max(output.data,
            return predicted

        def postprocess(self, data):
            return data.item()
```

In the preceding code, `initialize` is only called when the handler class is loaded. Here, you can load the model and all related functions (e.g., tokenizers).

The `preprocess` function allows you to preprocess the data. For example, if you want to classify text data, this function is the best place to convert your raw text into token IDs.

The `inference` function is where the actual inference happens. It is separate from the `preprocess` function, because you can change the device here.

And finally, `postprocess` allows you to process the predictions before they are returned to the API request. A good example is the conversion of class likelihoods into actual class labels.

The functions `preprocess`, `inference`, and `postprocess` are called with every model prediction request. You also only need to overwrite

the functions that need to be modified.

Once you have defined your handler, save it in the file we defined earlier in the model archive creation step (*/my_model/model.py*), and then create the archive.

TorchServe configuration

TorchServe lets you configure your deployment setup through a configuration file called *config.properties*. An example *config.properties* file could look like this:

```
# Basic configuration options
inference_address=http://0.0.0.0:8080
management_address=http://0.0.0.0:8081
metrics_address=http://0.0.0.0:8082
num_workers=2
model_store=/home/model_store/
install_py_dep_per_model=false
load_models=all
models=my_model.mar,another_model.mar

# Configure the logging location and level
log_location=/var/log/torchserve/
log_level=INFO
```

```
# Set the response timeout
default_response_timeout=120
```

The list of configuration options includes SSL support, gRPC ports, CORS configurations, and GPU configurations. The full list of configuration options is available on the PyTorch website.

On startup, TorchServe will try to locate the configuration file in the following order:

- If the environmental variable `TS_CONFIG_FILE` is set, TorchServe will use the provided path.
- If TorchServe was started with the `--ts-config` argument, the provided path will be used.
- If a `config.properties` file is in the current working directory, it will take the configuration from this file.

If none of the options are available, TorchServe will load a basic configuration with default values.

Making Model Prediction Requests

Your newly deployed PyTorch model can now be accessed through localhost. If you want to productize your model deployment, we highly

recommend that you deploy it via Kubernetes to assist with the scaling of the model inference loads:

```
$ curl -X POST http://127.0.0.1:8080/predictions/
```

Making Batch Inference Requests

The major benefit of deployment tools like TF Serving or TorchServe over plain Flask implementations is the capability of batch inferences. As we discussed in “Example: Deploying TensorFlow Models with TensorFlow Serving”, batch inferences let you use your underlying hardware more efficiently, and after some parameter tuning, you can increase your prediction throughput.

As with all model batch requests, the model server will either wait for a maximum time frame and infer the batch, or submit the batch as soon as it has reached its batch size limit.

TorchServe offers two ways to batch predictions. First, and very similar to TF Serving, it offers the same option to set up a global configuration file. Second, TorchServe allows you to submit the batch request settings with your inference request. This second option is very useful if you don’t want to update your configuration file and test a new batch configuration.

Setting batch configuration via config.properties

In your *config.properties* file you can replace the following line:

```
# Basic configuration options
...
models=my_model.mar,another_model.mar
...
```

with this batch configuration:

```
# Basic configuration options
...
models={
  "my_model": {
    "1.0": {
        "defaultVersion": true,
        "marName": "my_model.mar",
        "minWorkers": 1,
        "maxWorkers": 1,
        "batchSize": 4,
        "maxBatchDelay": 50,
        "responseTimeout": 120
    }
  },
  "another_model": {
    "1.0": {
```

```
            "defaultVersion": true,
            "marName": "another_model.mar",
            "minWorkers": 1,
            "maxWorkers": 4,
            "batchSize": 8,
            "maxBatchDelay": 100,
            "responseTimeout": 120
        }
    }
}
...
```

The configuration specifies the batch size and batch delay per model. That way, you can tune it specifically for each model and the production use cases.

Setting batch configuration via REST request

Alternatively, batch configurations can be set via TorchServe's model API. In the following POST request, the model `my_model` is registered with a batch size of 8 and a maximum batch delay of 50 ms:

```
$ curl -X POST
       "localhost:8081/models?url=my_model.mar&
```

Conclusion

In this chapter, we discussed how to deploy TensorFlow, JAX, and PyTorch ML models. We showed three hands-on examples. First, we introduced the deployment of JAX or TensorFlow models through TF Serving. Then, we demonstrated how to profile the serving performance. And lastly, we introduced how PyTorch models can be served with TorchServe.

You can find a full-color version of this plot online.

You can find a full-color version of this plot online.