simple-tensorflow-serving Documentation

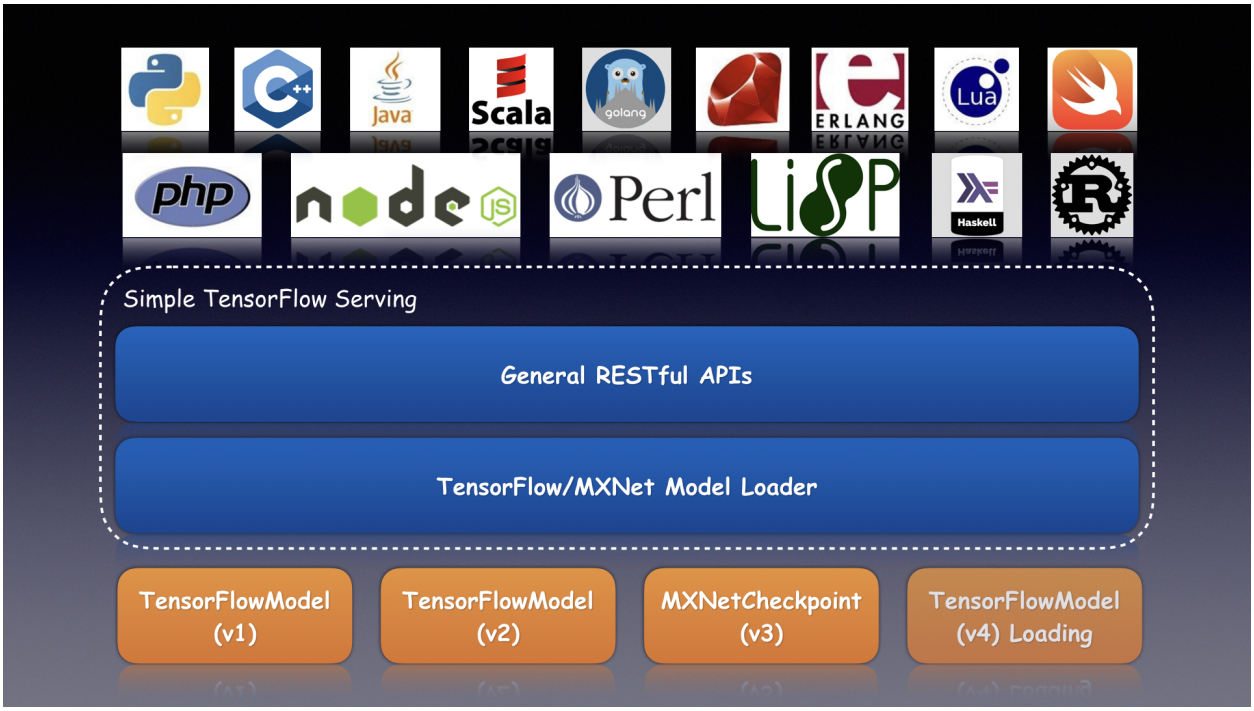
tobe

Nov 17, 2020

Contents:

CHAPTER 1

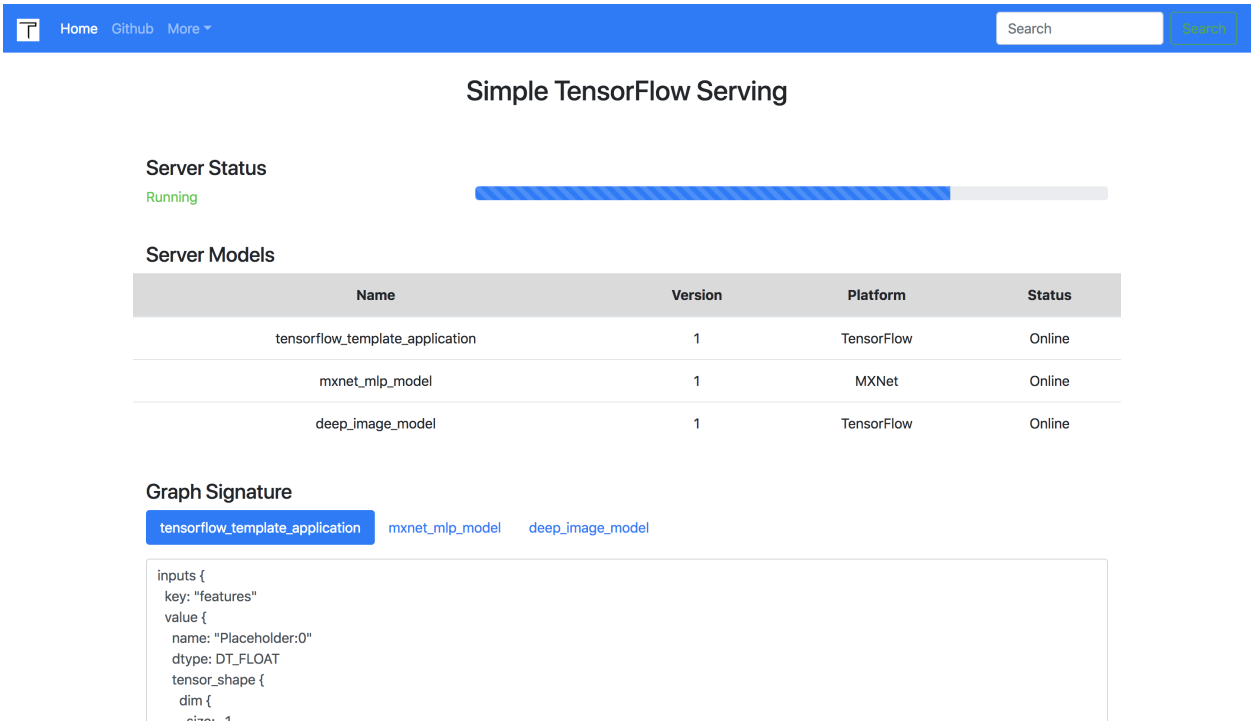
Introduction



Simple TensorFlow Serving is the generic and easy-to-use serving service for machine learning models.

- [x] Support distributed TensorFlow models
- [x] Support the general RESTful/HTTP APIs
- [x] Support inference with accelerated GPU
- [x] Support `curl` and other command-line tools
- [x] Support clients in any programming language
- [x] Support code-gen client by models without coding

- [x] Support inference with raw file for image models
- [x] Support statistical metrics for verbose requests
- [x] Support serving multiple models at the same time
- [x] Support dynamic online and offline for model versions
- [x] Support loading new custom op for TensorFlow models
- [x] Support secure authentication with configurable basic auth
- [x] Support multiple models of TensorFlow/MXNet/PyTorch/Caffe2/CNTK/ONNX/H2o/Scikit-learn/XGBoost/PMML



CHAPTER 2

Installation

2.1 Pip

Install the server with pip.

```
pip install simple_tensorflow_serving
```

2.2 Source

Install from source code.

```
git clone https://github.com/tobegit3hub/simple_tensorflow_serving

cd ./simple_tensorflow_serving/

python ./setup.py install
```

2.3 Bazel

Install with bazel.

```
git clone https://github.com/tobegit3hub/simple_tensorflow_serving

cd ./simple_tensorflow_serving/

bazel build simple_tensorflow_serving:server
```

2.4 Docker

Deploy with docker image.

```
docker run -d -p 8500:8500 tobegit3hub/simple_tensorflow_serving

docker run -d -p 8500:8500 tobegit3hub/simple_tensorflow_serving:latest-gpu

docker run -d -p 8500:8500 tobegit3hub/simple_tensorflow_serving:latest-hdfs

docker run -d -p 8500:8500 tobegit3hub/simple_tensorflow_serving:latest-py34
```

2.5 Docker Compose

Deploy with docker-compose.

```
wget https://raw.githubusercontent.com/tobegit3hub/simple_tensorflow_serving/master/
↪docker-compose.yml

docker-compose up -d
```

2.6 Kubernetes

Deploy in Kubernetes cluster.

```
wget https://raw.githubusercontent.com/tobegit3hub/simple_tensorflow_serving/master/
↪simple_tensorflow_serving.yaml

kubectl create -f ./simple_tensorflow_serving.yaml
```

CHAPTER 3

Quick Start

Train or download the TensorFlow SavedModel.

```
import tensorflow as tf

export_dir = "./model/1"
input_keys_placeholder = tf.placeholder(
    tf.int32, shape=[None, 1], name="input_keys")
output_keys = tf.identity(input_keys_placeholder, name="output_keys")

session = tf.Session()
tf.saved_model.simple_save(
    session,
    export_dir,
    inputs={"keys": input_keys_placeholder},
    outputs={"keys": output_keys})
```

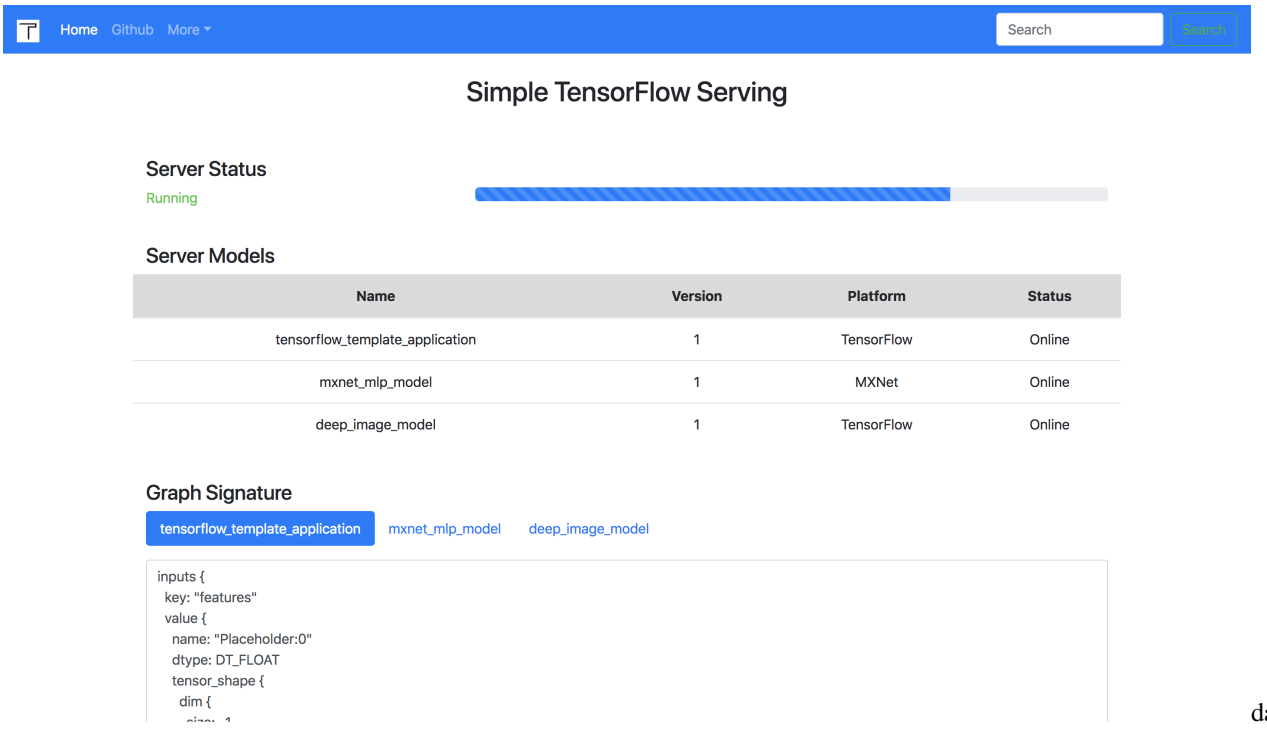
This script will export the model in `./model`.

```
$ tree ./model
./model
└── 1
    ├── saved_model.pb
    └── variables

2 directories, 1 file
```

Start serving to load the model.

```
simple_tensorflow_serving --model_base_path="./model"
```

Check out the dashboard in http://127.0.0.1:8500 in web browser.



dashboard

Generate the clients for testing without coding.

```
curl http://localhost:8500/v1/models/default/gen_client?language=python > client.py

python ./client.py
```

```
$ cat client.py
#!/usr/bin/env python

import requests

def main():
  endpoint = "http://127.0.0.1:8500"
  json_data = {"model_name": "default", "data": {"keys": [[1], [1]]} }
  result = requests.post(endpoint, json=json_data)
  print(result.json())

if __name__ == "__main__":
  main()
```

CHAPTER 4

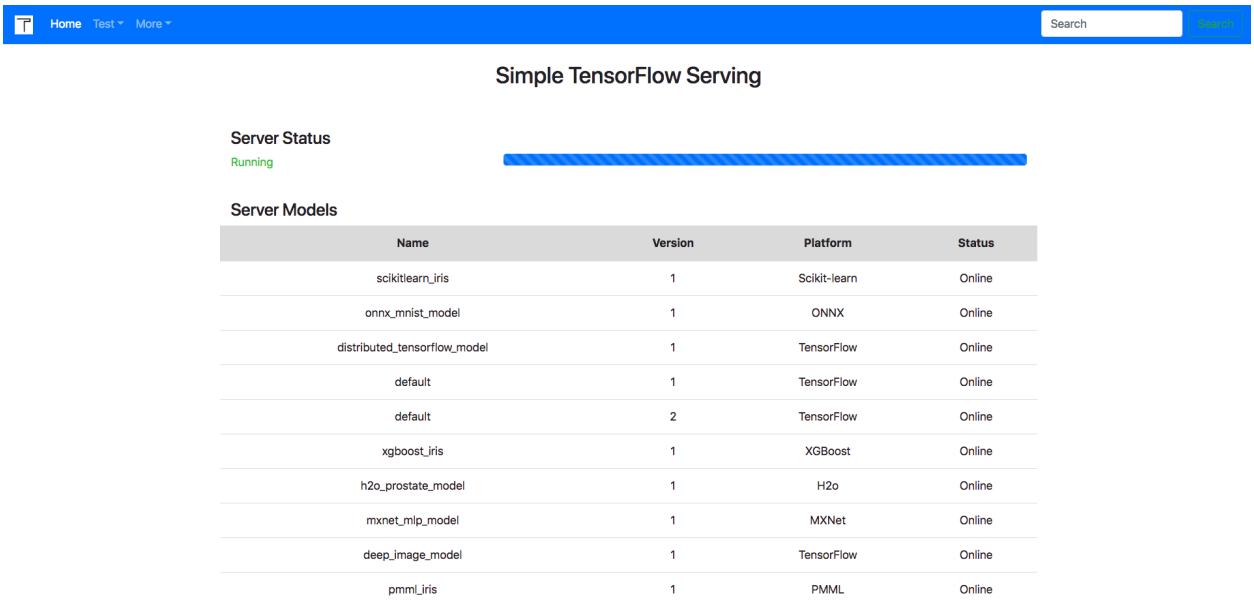
Advanced Usage

4.1 Multiple Models

It supports serve multiple models and multiple versions of these models. You can run the server with this configuration.

```
{
  "model_config_list": [
    {
      "name": "tensorflow_template_application_model",
      "base_path": "./models/tensorflow_template_application_model/",
      "platform": "tensorflow"
    }, {
      "name": "deep_image_model",
      "base_path": "./models/deep_image_model/",
      "platform": "tensorflow"
    }, {
       "name": "mxnet_mlp_model",
       "base_path": "./models/mxnet_mlp/mx_mlp",
       "platform": "mxnet"
    }
  ]
}
```

```
simple_tensorflow_serving --model_config_file="./examples/model_config_file.json"
```



Name	Version	Platform	Status
scikitlearn_iris	1	Scikit-learn	Online
onnx_mnist_model	1	ONNX	Online
distributed_tensorflow_model	1	TensorFlow	Online
default	1	TensorFlow	Online
default	2	TensorFlow	Online
xgboost_iris	1	XGBoost	Online
h2o_prostate_model	1	H2o	Online
mxnet_mlp_model	1	MXNet	Online
deep_image_model	1	TensorFlow	Online
pmml_iris	1	PMML	Online

4.2 GPU Acceleration

If you want to use GPU, try with the docker image with GPU tag and put cuda files in `/usr/cuda_files/`.

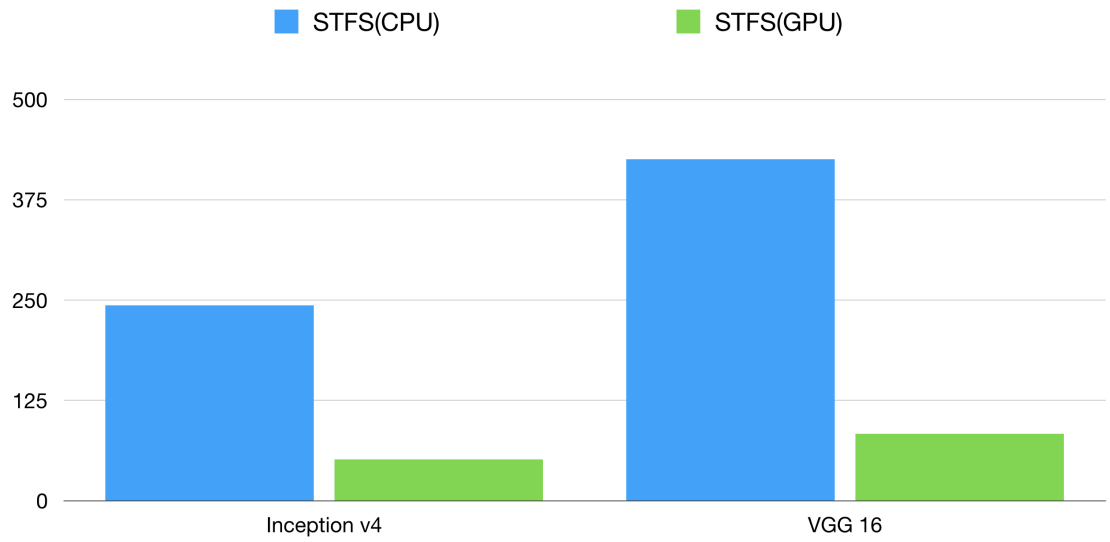
```
export CUDA_SO="-v /usr/cuda_files/:/usr/cuda_files/"
export DEVICES=$(\ls /dev/nvidia* | xargs -I{} echo '--device {}:{}')
export LIBRARY_ENV="-e LD_LIBRARY_PATH=/usr/local/cuda/extras/CUPTI/lib64:/usr/local/
↪nvidia/lib:/usr/local/nvidia/lib64:/usr/cuda_files"

docker run -it -p 8500:8500 $CUDA_SO $DEVICES $LIBRARY_ENV tobegit3hub/simple_
↪tensorflow_serving:latest-gpu
```

You can set session config and gpu options in command-line parameter or the model config file.

```
simple_tensorflow_serving --model_base_path="./models/tensorflow_template_application_
↪model" --session_config='{"log_device_placement": true, "allow_soft_placement":␣
↪true, "allow_growth": true, "per_process_gpu_memory_fraction": 0.5}'
```

```
{
  "model_config_list": [
    {
      "name": "default",
      "base_path": "./models/tensorflow_template_application_model/",
      "platform": "tensorflow",
      "session_config": {
        "log_device_placement": true,
        "allow_soft_placement": true,
        "allow_growth": true,
        "per_process_gpu_memory_fraction": 0.5
      }
    }
  ]
}
```

Here is the benchmark of CPU and GPU inference and y-coordinate is the latency(the lower the better).



4.3 Generated Client

You can generate the test json data for the online models.

```
curl http://localhost:8500/v1/models/default/gen_json
```

Or generate clients in different languages(Bash, Python, Golang, JavaScript etc.) for your model without writing any code.

```
curl http://localhost:8500/v1/models/default/gen_client?language=python > client.py
curl http://localhost:8500/v1/models/default/gen_client?language=bash > client.sh
curl http://localhost:8500/v1/models/default/gen_client?language=golang > client.go
curl http://localhost:8500/v1/models/default/gen_client?language=javascript > client.
↪js
```

The generated code should look like these which can be test immediately.

```
#!/usr/bin/env python

import requests

def main():
  endpoint = "http://127.0.0.1:8500"

  input_data = {"keys": [[1.0], [1.0]], "features": [[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.
↪0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]]}
  result = requests.post(endpoint, json=input_data)
  print(result.json())
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
  main()
```

4.4 Image Model

For image models, we can request with the raw image files instead of constructing array data.

Now start serving the image model like deep_image_model.

```
simple_tensorflow_serving --model_base_path="./models/deep_image_model/"
```

Then request with the raw image file which has the same shape of your model.

```
curl -X POST -F 'image=@./images/mew.jpg' -F "model_version=1" 127.0.0.1:8500
```

4.5 Custom Op

If your models rely on new TensorFlow custom op, you can run the server while loading the so files.

```
simple_tensorflow_serving --model_base_path="./model/" --custom_op_paths="./foo_op/"
```

Please check out the complete example in ./examples/custom_op/.

4.6 Authentication

For enterprises, we can enable basic auth for all the APIs and any anonymous request is denied.

Now start the server with the configured username and password.

```
./server.py --model_base_path="./models/tensorflow_template_application_model/" --
↪enable_auth=True --auth_username="admin" --auth_password="admin"
```

If you are using the Web dashboard, just type your certification. If you are using clients, give the username and password within the request.

```
curl -u admin:admin -H "Content-Type: application/json" -X POST -d '{"data": {"keys":␣
↪[[11.0], [2.0]], "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1,␣
↪1]]}}' http://127.0.0.1:8500
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "data": {
      "keys": [[11.0], [2.0]],
      "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]
  }
}
auth = requests.auth.HTTPBasicAuth("admin", "admin")
result = requests.post(endpoint, json=input_data, auth=auth)
```

4.7 TSL/SSL

It supports TSL/SSL and you can generate the self-signed secret files for testing.

```
openssl req -x509 -newkey rsa:4096 -nodes -out /tmp/secret.pem -keyout /tmp/secret.
↪key -days 365
```

Then run the server with certification files.

```
simple_tensorflow_serving --enable_ssl=True --secret_pem=/tmp/secret.pem --secret_
↪key=/tmp/secret.key --model_base_path="./models/tensorflow_template_application_
↪model"
```

CHAPTER 5

API

5.1 RESTful API

The most import API is inference for the loaded models.

```
Endpoint: /
Method: POST
JSON: {
        "model_name": "default",
        "model_version": 1,
        "data": {
            "keys": [[11.0], [2.0]],
            "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1],
                         [1, 1, 1, 1, 1, 1, 1, 1, 1]]
        }
      }
Response: {
            "keys": [[1], [1]]
          }
```

5.2 Python Example

You can easily choose the specified model and version for inference.

```
import requests

endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
```

(continues on next page)

(continued from previous page)

```
      "keys": [[11.0], [2.0]],
      "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1, 1, 1, 1, 1]]
  }
}
result = requests.post(endpoint, json=input_data)
```

CHAPTER 6

Models

6.1 TensorFlow

For TensorFlow models, you can load with commands and configuration like these.

```
simple_tensorflow_serving --model_base_path="./models/tensorflow_template_application_
↪model" --model_platform="tensorflow"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[12.0, 2.0]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.2 MXNET

For MXNet models, you can load with commands and configuration like these.

```
simple_tensorflow_serving --model_base_path="./models/mxnet_mlp/mx_mlp" --model_
↪platform="mxnet"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
```

(continues on next page)

(continued from previous page)

```
      "data": [[12.0, 2.0]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.3 ONNX

For ONNX models, you can load with commands and configuration like these.

```
simple_tensorflow_serving --model_base_path="./models/onnx_mnist_model/onnx_model.
↪proto" --model_platform="onnx"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[...]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.4 Scikit-learn

For Scikit-learn models, you can load with commands and configuration like these.

```
simple_tensorflow_serving --model_base_path="./models/scikitlearn_iris/model.joblib" -
↪-model_platform="scikitlearn"

simple_tensorflow_serving --model_base_path="./models/scikitlearn_iris/model.pkl" --
↪model_platform="scikitlearn"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[...]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.5 XGBoost

For XGBoost models, you can load with commands and configuration like these.

```
simple_tensorflow_serving --model_base_path="./models/xgboost_iris/model.bst" --model_
↪platform="xgboost"

simple_tensorflow_serving --model_base_path="./models/xgboost_iris/model.joblib" --
↪model_platform="xgboost"

simple_tensorflow_serving --model_base_path="./models/xgboost_iris/model.pkl" --model_
↪platform="xgboost"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[...]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.6 PMML

For PMML models, you can load with commands and configuration like these. This relies on Openscoring and Openscoring-Python to load the models.

```
java -jar ./third_party/openscoring/openscoring-server-executable-1.4-SNAPSHOT.jar

simple_tensorflow_serving --model_base_path="./models/pmml_iris/DecisionTreeIris.pmml
↪" --model_platform="pmml"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[...]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

6.7 H2o

For H2o models, you can load with commands and configuration like these.

```
# Start H2o server with "java -jar h2o.jar"

simple_tensorflow_serving --model_base_path="./models/h2o_prostate_model/GLM_model_
↪python_1525255083960_17" --model_platform="h2o"
```

```
endpoint = "http://127.0.0.1:8500"
input_data = {
  "model_name": "default",
  "model_version": 1,
  "data": {
      "data": [[...]]
  }
}
result = requests.post(endpoint, json=input_data)
print(result.text)
```

CHAPTER 7

Clients

7.1 Bash

Here is the example client in Bash.

```
curl -H "Content-Type: application/json" -X POST -d '{"data": {"keys": [[1.0], [2.0]],
↪ "features": [[10, 10, 10, 8, 6, 1, 8, 9, 1], [6, 2, 1, 1, 1, 1, 7, 1, 1]]}}' http:/
↪/127.0.0.1:8500
```

7.2 Python

Here is the example client in Python.

```
endpoint = "http://127.0.0.1:8500"
payload = {"data": {"keys": [[11.0], [2.0]], "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1],
↪ [1, 1, 1, 1, 1, 1, 1, 1, 1]]}}

result = requests.post(endpoint, json=payload)
```

7.3 Golang

```
endpoint := "http://127.0.0.1:8500"
dataByte := []byte(`{"data": {"keys": [[11.0], [2.0]], "features": [[1, 1, 1, 1, 1, 1,
↪ 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]}}`)
var dataInterface map[string]interface{}
json.Unmarshal(dataByte, &dataInterface)
dataJson, _ := json.Marshal(dataInterface)

resp, err := http.Post(endpoint, "application/json", bytes.NewBuffer(dataJson))
```

7.4 Ruby

Here is the example client in Ruby.

```
endpoint = "http://127.0.0.1:8500"
uri = URI.parse(endpoint)
header = {"Content-Type" => "application/json"}
input_data = {"data" => {"keys"=> [[11.0], [2.0]], "features"=> [[1, 1, 1, 1, 1, 1, 1,
↪ 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]}}
http = Net::HTTP.new(uri.host, uri.port)
request = Net::HTTP::Post.new(uri.request_uri, header)
request.body = input_data.to_json

response = http.request(request)
```

7.5 JavaScript

Here is the example client in JavaScript.

```
var options = {
    uri: "http://127.0.0.1:8500",
    method: "POST",
    json: {"data": {"keys": [[11.0], [2.0]], "features": [[1, 1, 1, 1, 1, 1, 1, 1, 1],
↪ [1, 1, 1, 1, 1, 1, 1, 1, 1]]}}
};

request(options, function (error, response, body) {});
```

7.6 PHP

Here is the example client in PHP.

```
$endpoint = "127.0.0.1:8500";
$inputData = array(
    "keys" => [[11.0], [2.0]],
    "features" => [[1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]],
);
$jsonData = array(
    "data" => $inputData,
);
$ch = curl_init($endpoint);
curl_setopt_array($ch, array(
    CURLOPT_POST => TRUE,
    CURLOPT_RETURNTRANSFER => TRUE,
    CURLOPT_HTTPHEADER => array(
        "Content-Type: application/json"
    ),
    CURLOPT_POSTFIELDS => json_encode($jsonData)
));

$response = curl_exec($ch);
```

7.7 Erlang

Here is the example client in Erlang.

```
ssl:start(),
application:start(inets),
httpc:request(post,
  {"http://127.0.0.1:8500", [],
  "application/json",
  "{\"data\": {\"keys\": [[11.0], [2.0]], \"features\": [[1, 1, 1, 1, 1, 1, 1, 1, 1],
↪[1, 1, 1, 1, 1, 1, 1, 1, 1]]}}"
  }, [], []).
```

7.8 Lua

Here is the example client in Lua.

```
local endpoint = "http://127.0.0.1:8500"
keys_array = {}
keys_array[1] = {1.0}
keys_array[2] = {2.0}
features_array = {}
features_array[1] = {1, 1, 1, 1, 1, 1, 1, 1, 1}
features_array[2] = {1, 1, 1, 1, 1, 1, 1, 1, 1}
local input_data = {
    ["keys"] = keys_array,
    ["features"] = features_array,
}
local json_data = {
    ["data"] = input_data
}
request_body = json:encode (json_data)
local response_body = {}

local res, code, response_headers = http.request{
    url = endpoint,
    method = "POST",
    headers =
      {
          ["Content-Type"] = "application/json";
          ["Content-Length"] = #request_body;
      },
      source = ltn12.source.string(request_body),
      sink = ltn12.sink.table(response_body),
}
```

7.9 Perl

Here is the example client in Perl.

```
my $endpoint = "http://127.0.0.1:8500";
my $json = '{"data": {"keys": [[11.0], [2.0]], "features": [[1, 1, 1, 1, 1, 1, 1, 1,
↪1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]}}';
```

(continues on next page)

(continued from previous page)

```
my $req = HTTP::Request->new( 'POST', $endpoint );
$req->header( 'Content-Type' => 'application/json' );
$req->content( $json );
$ua = LWP::UserAgent->new;

$response = $ua->request($req);
```

7.10 R

Here is the example client in R.

```
endpoint <- "http://127.0.0.1:8500"
body <- list(data = list(a = 1), keys = 1)
json_data <- list(
  data = list(
    keys = list(list(1.0), list(2.0)), features = list(list(1, 1, 1, 1, 1, 1, 1, 1,
↪1), list(1, 1, 1, 1, 1, 1, 1, 1, 1))
  )
)

r <- POST(endpoint, body = json_data, encode = "json")
stop_for_status(r)
content(r, "parsed", "text/html")
```

7.11 Postman

Here is the example with Postman.

CHAPTER 8

Image Model

8.1 Introduction

Simple TensorFlow Serving has extra support for image models. You can deploy the image models easily and make inferences by uploading the image files in web browser or using form-data. The best practice is accepting base64 strings as input of model signature like this.

```
inputs {
  key: "images"
  value {
    name: "model_input_b64_images:0"
    dtype: DT_STRING
    tensor_shape {
      dim {
        size: -1
      }
    }
  }
}
```

8.2 Export Image Model

Model images should be standard TensorFlow SavedModel as well. We do not use `[batch_size, r, g, b]` or `[batch_size, r, b, g]` as signature input because it is not compatible with arbitrary image files. We can accept the base64 strings as input, then decode and resize the tensor for the required model input.

```
# Define model
def inference(input):
  weights = tf.get_variable(
      "weights", [784, 10], initializer=tf.random_normal_initializer())
  bias = tf.get_variable(
```

(continues on next page)

(continued from previous page)

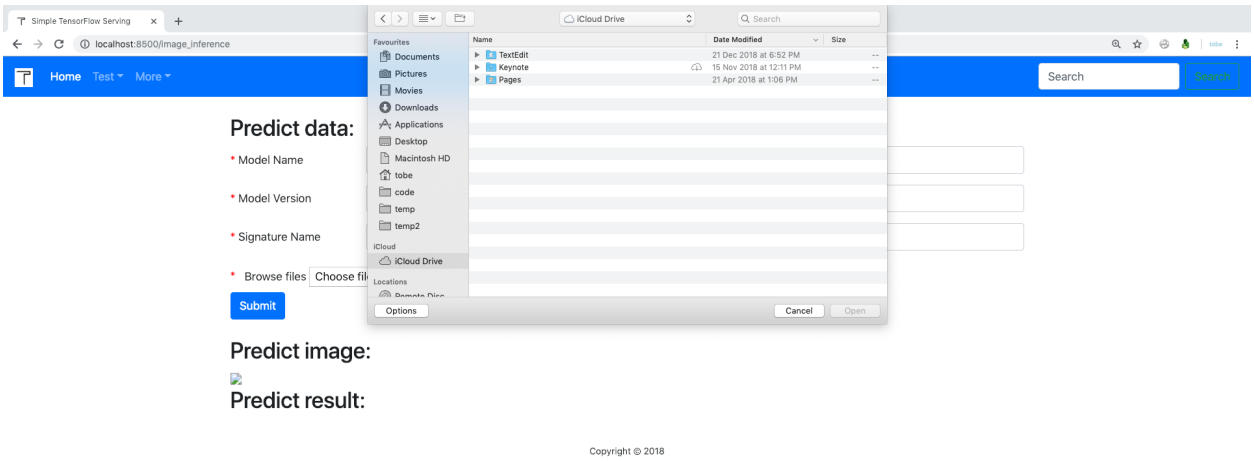
```
      "bias", [10], initializer=tf.random_normal_initializer())
  logits = tf.matmul(input, weights) + bias

  return logits


# Define op for model signature
tf.get_variable_scope().reuse_variables()

model_base64_placeholder = tf.placeholder(
    shape=[None], dtype=tf.string, name="model_input_b64_images")
model_base64_string = tf.decode_base64(model_base64_placeholder)
model_base64_input = tf.map_fn(lambda x: tf.image.resize_images(tf.image.decode_
↪jpeg(x, channels=1), [28, 28]), model_base64_string, dtype=tf.float32)
model_base64_reshape_input = tf.reshape(model_base64_input, [-1, 28 * 28])
model_logits = inference(model_base64_reshape_input)
model_predict_softmax = tf.nn.softmax(model_logits)
model_predict = tf.argmax(model_predict_softmax, 1)


# Export model
export_dir = "./model/1"
tf.saved_model.simple_save(
    sess,
    export_dir,
    inputs={"images": model_base64_placeholder},
    outputs={
        "predict": model_predict,
        "probability": model_predict_softmax
    })
```

8.3 Inference With Uploaded Files

Now we can start Simple TensorFlow Serving and load the image models easily. Take the deep_image_model for example.

```
git clone https://github.com/tobegit3hub/simple_tensorflow_serving

cd ./simple_tensorflow_serving/models/

simple_tensorflow_serving --model_base_path="./deep_image_model"
```

Then you can choose the local image file to make inference.



8.4 Inference with Python Client

If you want to make inferences with Python client. You can encode the image file with the base64 library.

```
import requests
import base64

def main():
  image_string = base64.urlsafe_b64encode(open("./test.png", "rb").read())

  endpoint = "http://127.0.0.1:8500"
  json_data = {"model_name": "default", "data": {"images": [image_string]} }
  result = requests.post(endpoint, json=json_data)
  print(result.json())

if __name__ == "__main__":
  main()
```

Here is the exmaple data of one image's base64 string.

```
{"images": ["_9j_4AAQSkZJRgABAQAASABIAAD_
↪4QCMRXhpZgAATU0AKgAAAAgABQESAAMAAAABAAEAAAEaAAUAAAABAAAASgEbAAUAAAABAAAAUgEoAAMAAAABAAIAAIdpAAQAAA
↪-
↪EJIWh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8APD94cGFja2V0IGJlZ2luPSLvu78iIGlkPSJXNU0wTXBDZWhpSHpyZVN6T
↪IDwvcmRmOlJERj4gPC94OnhtcG1ldGE-
↪ICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICA
↪PgD_7QA4UGhvdG9zaG9wIDMuMAA4QklNBAQAAAAAAAA4QklNBCUAAAAAABDUHYzZjwCyBOmACZjs-EJ-_
↪8AAEQgAIAAgAwEiAAIRAQMRAf_EAB8AAAEFAQEBAQEBAAAAAAAAAAABAgMEBQYHCAkKC__
↪EALUQAAIBAwMCBAMFBQQEAAABfQECAwAEEQUSITFBBhNRYQcicRQygZGhCCNCscEVUtHwJDNicoIJChYXGBkaJSYnKCkqNDU2Nz
↪j5-v_EAB8BAAMBAQEBAQEBAQEAAAAAAAABAgMEBQYHCAkKC__
↪EALURAAIBAgQEAwQHBQQEAAECdwABAgMRBAUhMQYSQVEHYXETIjKBCBRCkaGxwQkjM1LwFWJy0QoWJDThJfEXGBkaJicoKSo1Nj
↪jp6vLz9PX29_j5-v_
↪bAEMAAgICAgICAwICAwUDAwMFBgUFBQUGCAYGBgYGCAoICAgICAgKCgoKCgoKCgwMDAwMDA4ODg4ODw8PDw8PDw8PD_
↪_
↪bAEMBAgICBAQEBwQEBxALCQsQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEP_
↪dAAQAAv_aAAwDAQACEQMRAD8A_bzx547t_CFokNtH9s1W6B-z2wySe29gOdoPpye2ACRwWg-ItctVm1_
↪xXHCbxSUhUSkgI33m4BHsAMADnnPGz4c8S-
↪EdOW71nX9Sgt9e1F2NykjbbiFVYiO3VCN4Ea4GAPmbLc5zXkus3UtnfXbxW99fQtHCbZZbcxiTYu0sqPtdVc4OWUDrg4FW1ZWPc
↪uJe-Hjb2mm69f32tXzKtrbSmGQPgje7oI12xKMl2yMdAdxAK5ddDzsXheWq4bH_9D9-ti7t-BuHfvXJ-
↪Nbyz0nw5falckQoBHHJJjkI8iocn0_b8K66snXdJh1zSLrSp_FuUK564PVT
↪BANBrQmozTe1z5mm8Q3UTzajoy3aYt2uI7mCAyKmR1JYbADwfm-UjrXvHgjw_o-m6cmr2dxJqd5qcccs-
↪oXDb57jIyMnoij-GNAEXoBU-
↪[illegible]KWIGcLx06cVF4Jvo57GW0jdZPszDIXGELjLIQOhVs8dQCMintoepmGL9[illegible]La2nqj_
↪_2Q=="]}
```

(continues on next page)

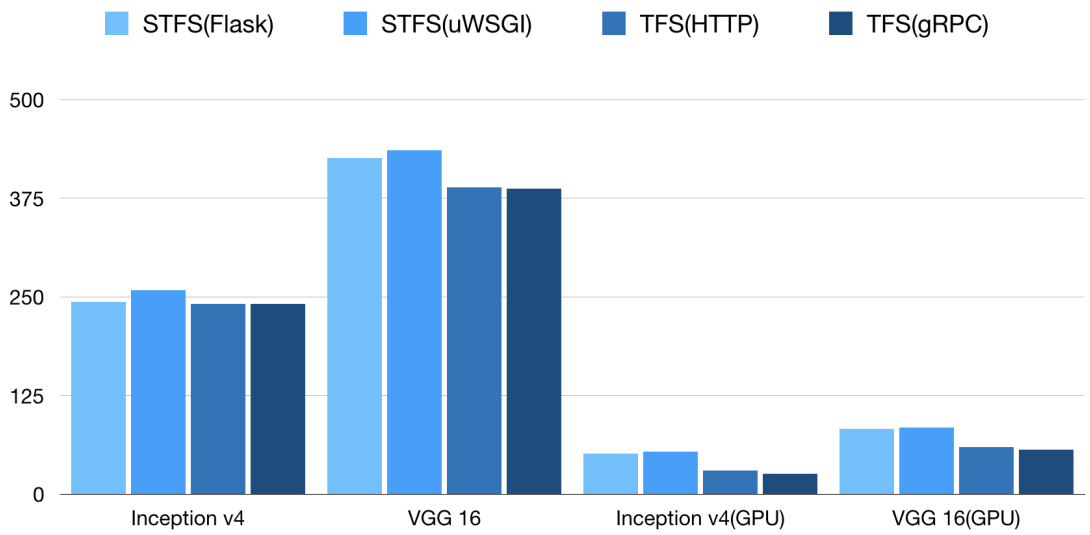
(continued from previous page)

CHAPTER 9

Performance

You can run SimpleTensorFlowServing with any WSGI server for better performance. We have benchmarked and compare with `TensorFlow Serving`. Find more details in benchmark directory.

STFS(Simple TensorFlow Serving) and TFS(TensorFlow Serving) have similar performances for different models. Vertical coordinate is inference latency(microsecond) and the less is better.



Then we test with `ab` with concurrent clients in CPU and GPU. `TensorFlow Serving` works better especially with GPUs.

	Flask	uWSGI	TensorFlow Serving
Inception / c1	251.851	262.203	251.036
Inception / c10	260.53	255.711	251.036
Inception / c100	251.042	254.486	211.876
VGG 16 / c1	428.767	440.099	426.314
VGG 16 / c10	358.162	360.651	336.915
VGG 16 / c100	347.316	350.416	334.372
Inception(GPU) / c1	53.203	58.34	32.148
Inception(GPU) / c10	35.779	38.773	24.716
Inception(GPU) / c100	45.092	36.635	25.61
VGG(GPU) / c1	84.181	83.317	59.233
VGG(GPU) / c10	54.532	53.799	53.16
VGG(GPU) / c100	57.025	53.748	53.051

For simplest model, each request only costs ~1.9 microseconds and one instance of Simple TensorFlow Serving can achieve 5000+ QPS. With larger batch size, it can inference more than 1M instances per second.

Batch size	1	16	128	1024	8192	65536	524288	4194304
Latency (ms)	5.483	5.085	5.092	5.928	14.358	77.520	605.725	3959.240
Requests per second	182.382	196.657	196.386	168.811	69.648	12.900	1.651	0.253
Instances per second	182.380	3146.512	25137.408	172862.464	570556.416	845414.400	865599.488	1061158.912
Accelerate ratio	1.0	17.253↑	7.989↑	6.877↑	3.301↑	1.481↑	1.024↑	1.226↑

CHAPTER 10

Development

10.1 Principle

1. `simple_tensorflow_serving` starts the HTTP server with `flask` application.
2. Load the TensorFlow models with `tf.saved_model.loader` Python API.
3. Construct the feed_dict data from the JSON body of the request.

```
// Method: POST, Content-Type: application/json
{
  "model_version": 1, // Optional
  "data": {
    "keys": [[1], [2]],
    "features": [[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 2.0, 3.0, 4.
↪0, 5.0, 6.0, 7.0, 8.0, 9.0]]
  }
}
```

4. Use the TensorFlow Python API to `sess.run()` with feed_dict data.
5. For multiple versions supported, it starts independent thread to load models.
6. For generated clients, it reads user's model and render code with Jinja templates.

10.2 Debug

You can install the server with `develop` and test when code changes.

```
git clone https://github.com/tobegit3hub/simple_tensorflow_serving

cd ./simple_tensorflow_serving/

python ./setup.py develop
```