Deploying Models at Scale

TF Serving, TF Lite, and Google Cloud Platform

N. Rich Nguyen, PhD

SYS 6016

Most ML models never got deployed :(

Deployment is (used to be) hard

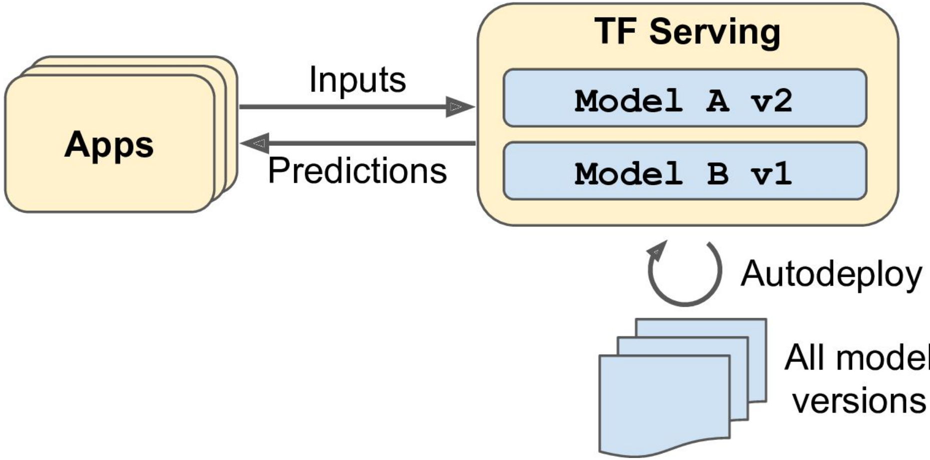
until...TensorFlow Serving

+

TensorFlow Serving

A production ready framework to serve ML models

- Part of the TensorFlow Extended Ecosystem
- Battle-tested: **used internally** at Google
- Highly scalable model serving solution
- Works well for large models up to 2GB
- Checks for new model versions



Export your Model

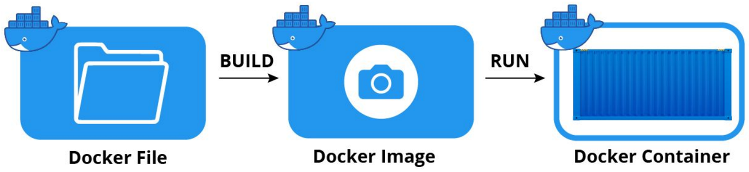
```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

- TensorFlow SavedModel Export
- Consistent model export
- Using Protobuf format (*saved_model.pb*)
- Variables and checkpoints
- Assets contains additional files

Installing TensorFlow Serving

1. First install **Docker** (https://docker.com)
2. Download **TF Serving Docker Image**: it's simple to install, does not mess with your system, and offers high performance.
3. Create a **Docker Container** to run this image



interactive output

delete container when stop

serve gRPC

serve REST

make host directory available to container

set container an environment variable

name of image

```
$ docker pull tensorflow/serving

$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
    -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
    -e MODEL_NAME=my_mnist_model \
    tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

Querying TF Serving through REST API

Creating a query with JSON

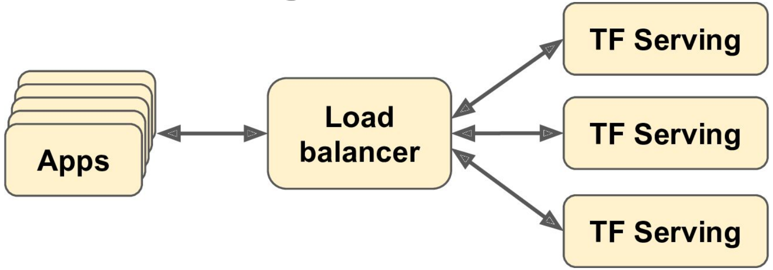
```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

Send the input data to TF Serving by sending an HTTP POST request

```
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()

>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

Scaling up TF Serving

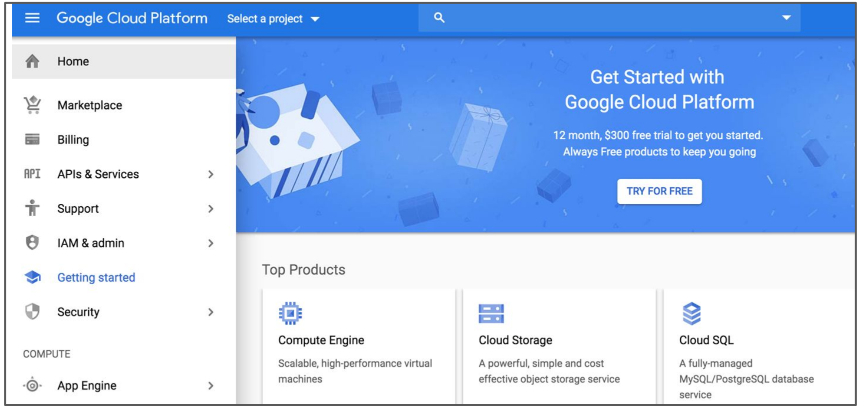


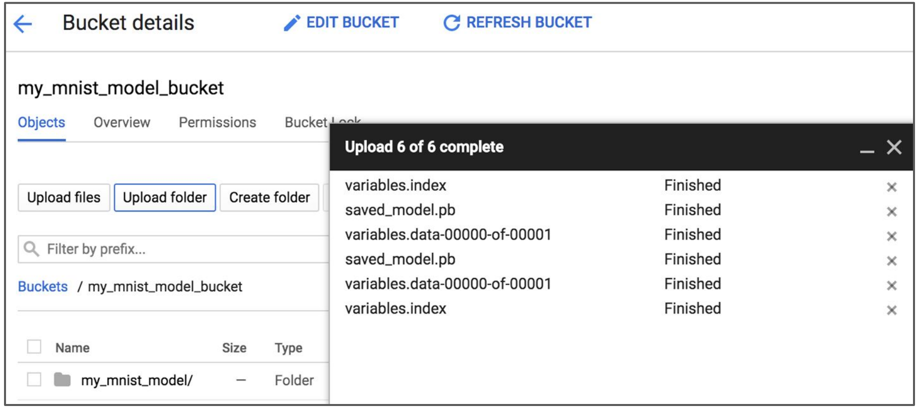
- If you expect to have many queries per second, deploy TF Serving on multiple servers and load-balanced the queries
- Purchase, maintain, and upgrade hardware infrastructures → not a way to go!
- Use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Oracle Cloud, ect.
- Managing all virtual machines, handling containers, configuration, tuning, and monitoring --- all of this can be a full-time job :(
- **Google Cloud Platform** provides many benefits: offers TPU computation, supports TensorFlow, has a nice suite of AI services (AutoML,Vision,NLP API)

Google Cloud Platform

Logging in account and console

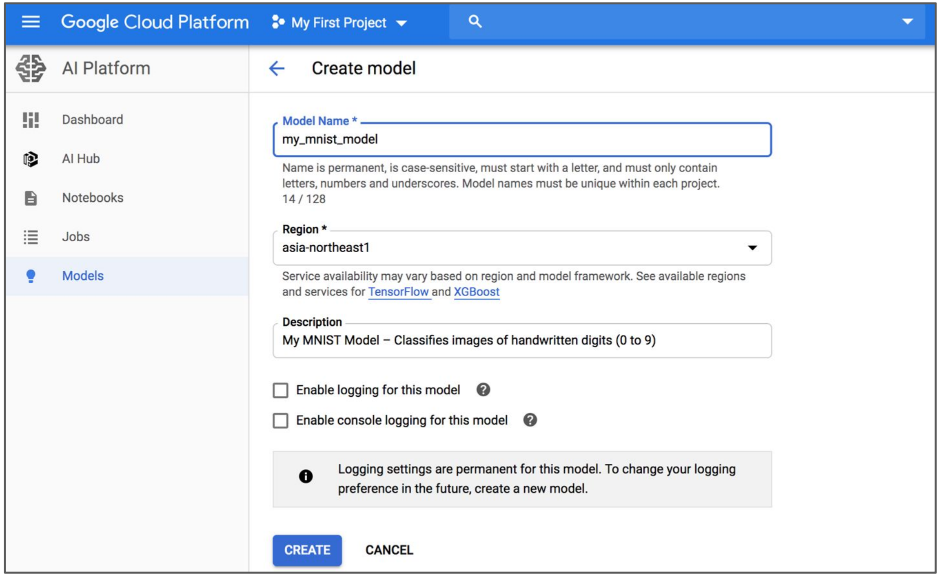
- Log in your Google account
- Go to console.cloud.google.com
- Keep in mind, this service is **not free**, but you can start with free credits
- Resources in Google Cloud Platform (GCP) belongs to a **project**
- You first need Google Cloud Storage (GCS) → create a **bucket**

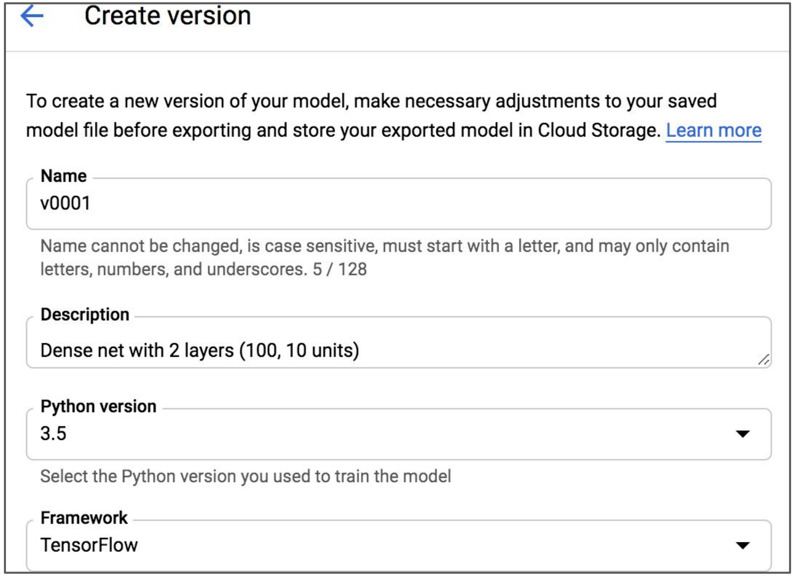




Uploading your model

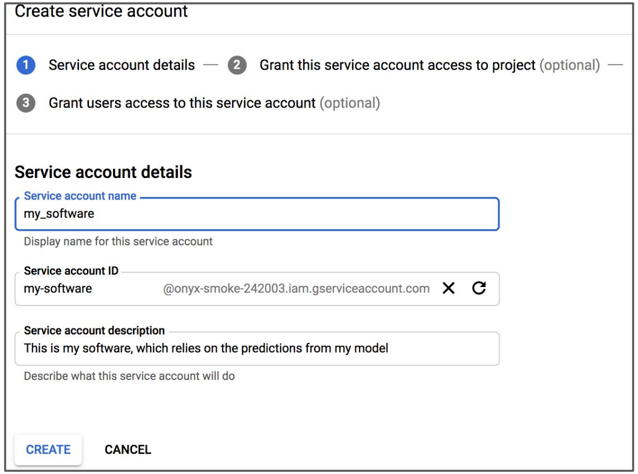
- Upload the *my_mnist_model* folder to your **bucket**
- Configure your AI Platform so it knows which models to use
- Select version, framework, runtime, machine type, model path.





Using the Prediction Service

- GCP runs TF Serving, and takes care of encryption and authentication
- Encryption is based on SSL/TLS, and authentication is **token-based**
- The client code can authenticate with **a service account** representing an application, not a user.
- It will get very restricted access rights: just what it needs and no more



Writing a script to query the prediction service

```python
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

```python
import googleapiclient.discovery

project_id = "onyx-smoke-242003" # change this to your project ID
model_id = "my_mnist_model"
model_path = "projects/{}/models/{}".format(project_id, model_id)
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

```python
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                       "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name] for pred in response["predictions"]])
```

```python
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

Deploying to Mobile Devices

Limitations of mobile devices

Large model may simply take too long to download, use too much RAM and CPU → make your app unresponsive, heat the device, and drain its battery :(

We need to make a mobile-friendly, lightweight, and efficient model w/o dropping too much accuracy...

TFLite library provides tools to help:

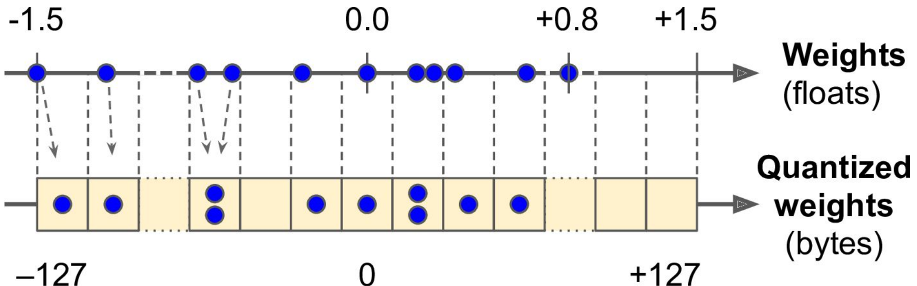
- Reduce model size
- Reduce the computational needed for prediction
- Adapt model to device specific constraints

TFLite Converter

- TFLite can take a ***SavedModel*** and compress it to a much lighter format based on **FlatBuffers**, an efficient serialization library (similar to ProtoBuf)
- FlatBuffer model (saved as a `.tflite` file) can be loaded directly to RAM without any preprocessing → reduce loading time and memory footprint.

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```

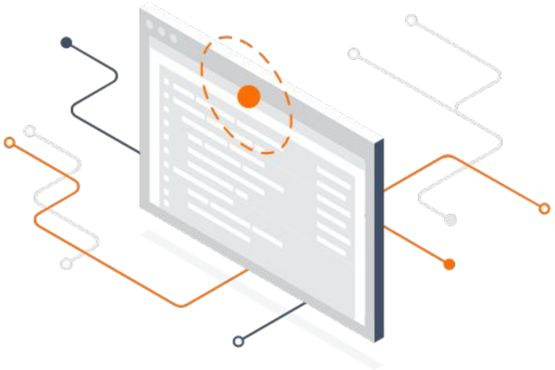
- Another way to reduce the model size is **quantizing** the model weights down to smaller bit-widths: 32 bit floats → 16 bit floats → 8 bit integers!



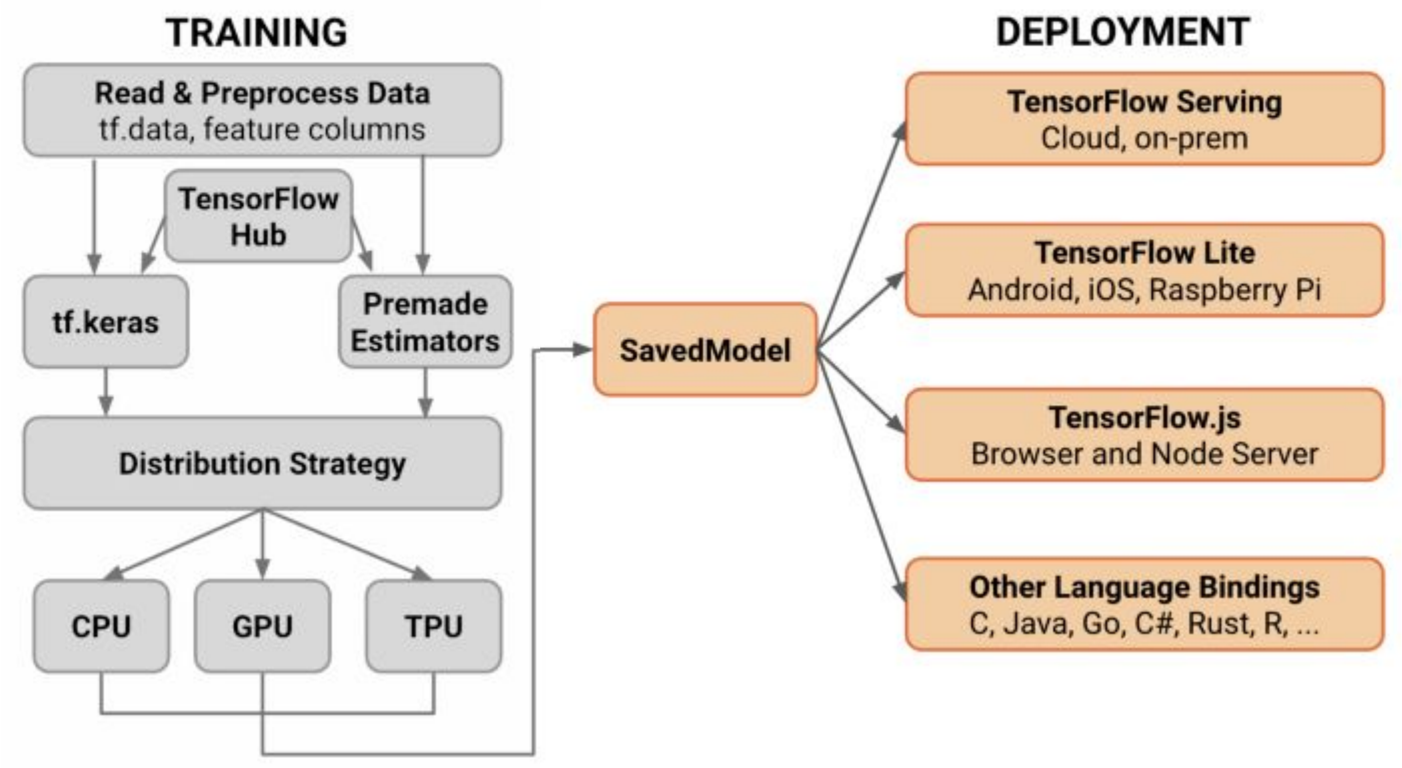
TensorFlow in the Browser

- Use model in a website → run it directly in the user's browser
  - Make your website more reliable
  - Remove the need to query server to make prediction
  - Reduce latency and make website more responsive
- Export your model to a special format that can be loaded by the **TensorFlow.js** Javascript library.

→ **The topic of our next video!**



Summary: TensorFlow EcoSystem



Bonus Content

Querying TF Serving through gRPC API

Serialize PredictRequest protobuf as input

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

Send the request to the server and get its response

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```