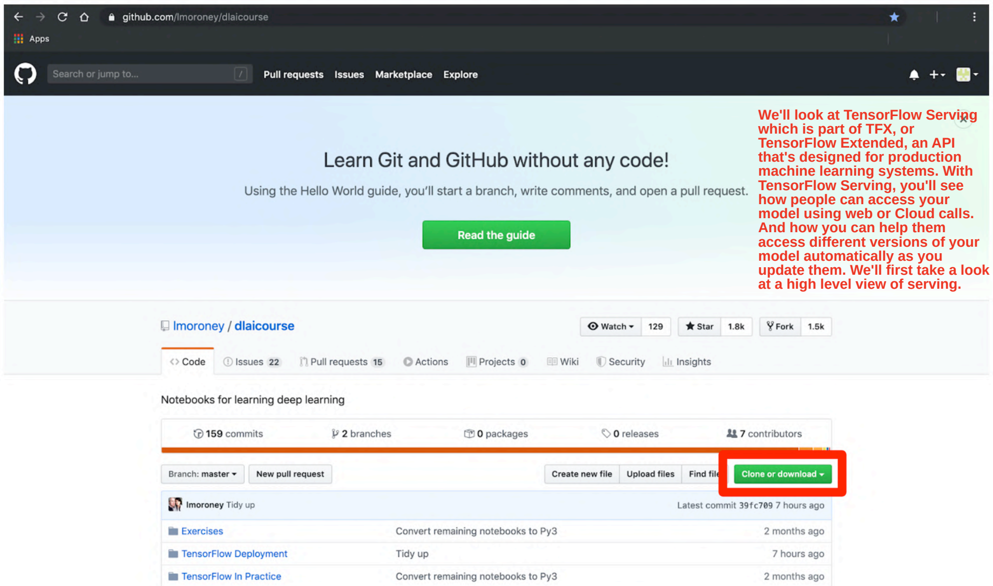
Downloading the Coding Examples and Exercises

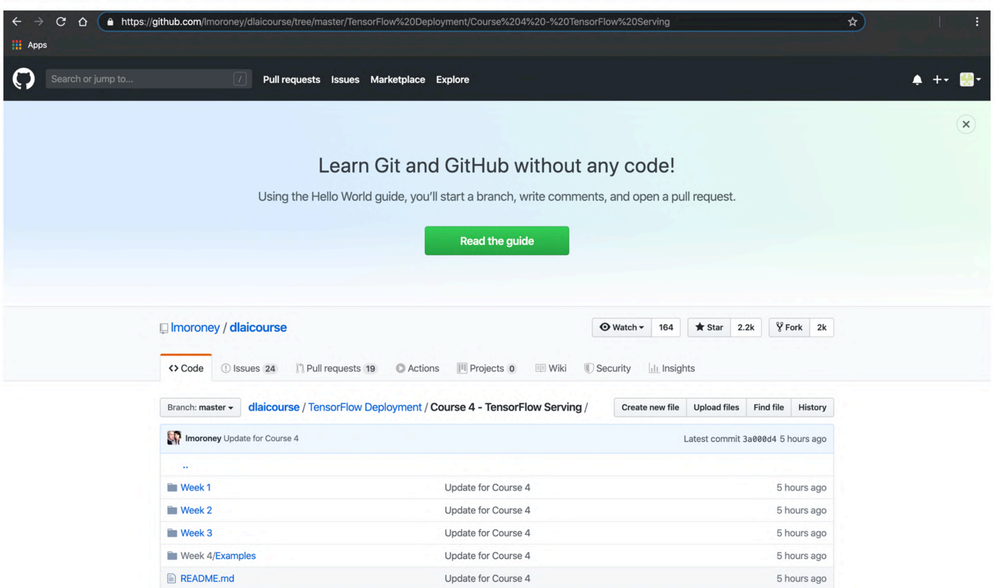
We have created this [GitHub Repository]() where you can find all the examples and exercises not only for this course but for the entire TensorFlow for Data and Deployment Specialization .

You can download all the examples and exercises to your computer by cloning or downloading the GitHub Repository.



You can find the corresponding coding examples and exercises for this course in the following folder in the GitHub repository:

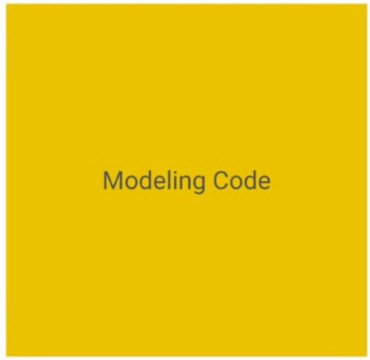
[dlaicourse/TensorFlow Deployment/Course 4 - TensorFlow Serving/]()



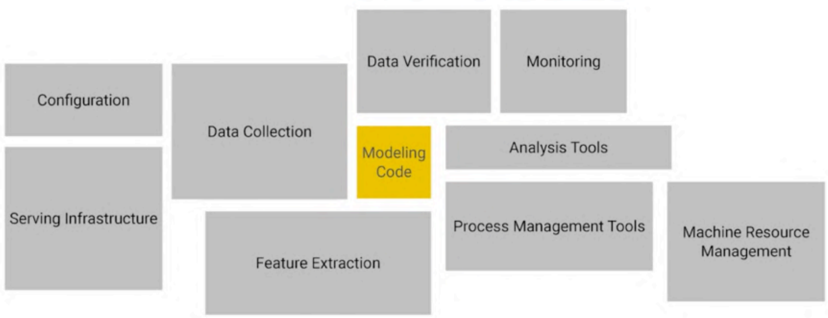
Each folder contains the corresponding examples and exercises for each week of this course on TensorFlow Serving.

NOTE: The code in the repository is updated occasionally. Therefore the code in the repository may vary slightly from the one shown in the videos.

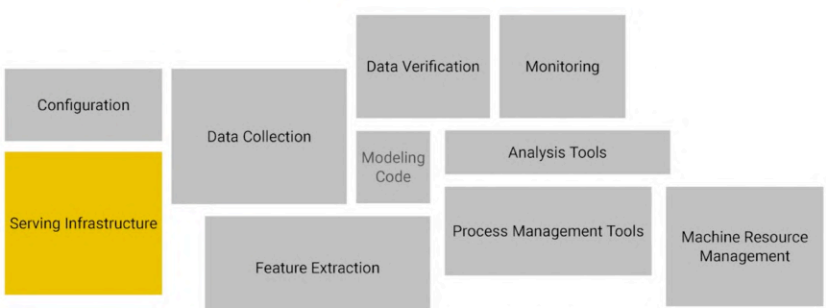
Building Models is just a small part of ML...

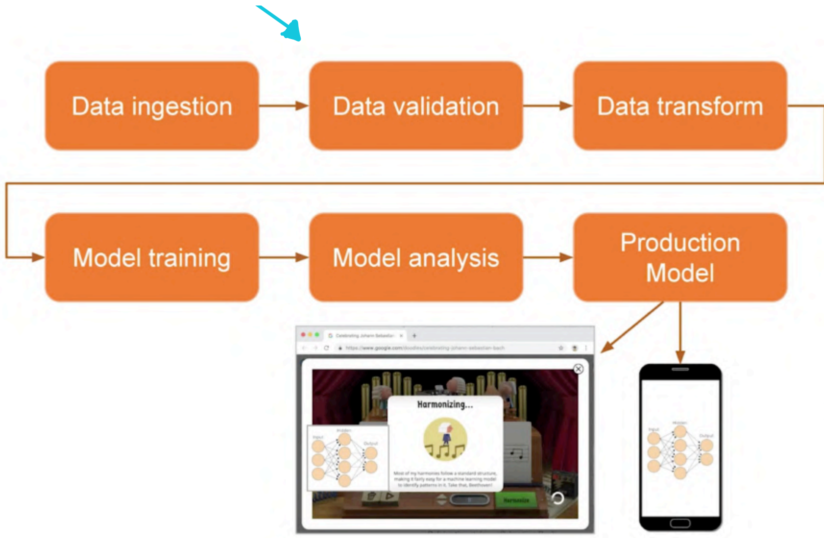


... a production solution requires so much more

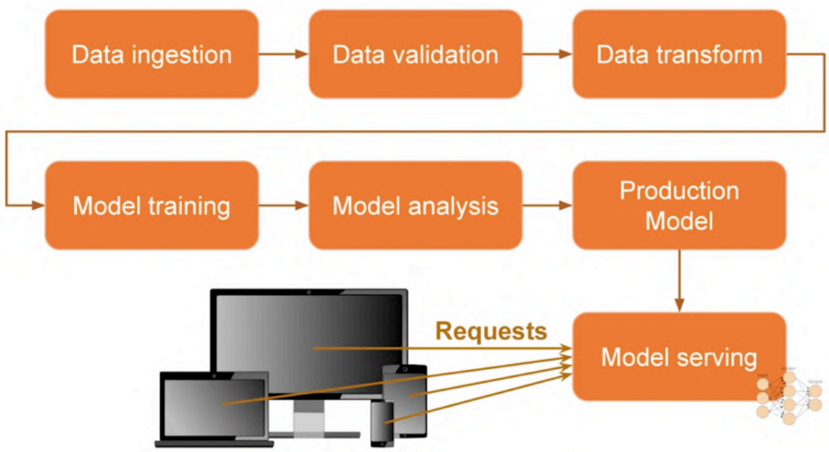


So this week we will focus on Serving...

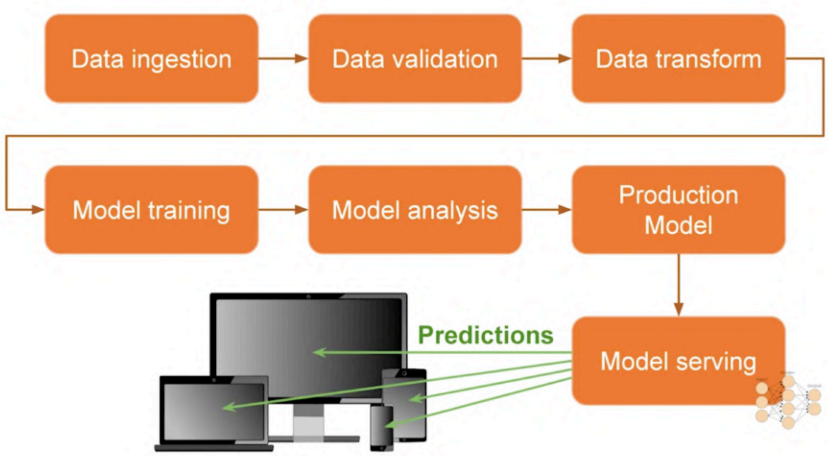




In both of these cases, the model was deployed to a remote device, be it native mobile or a browser based on the mobile or desktop.

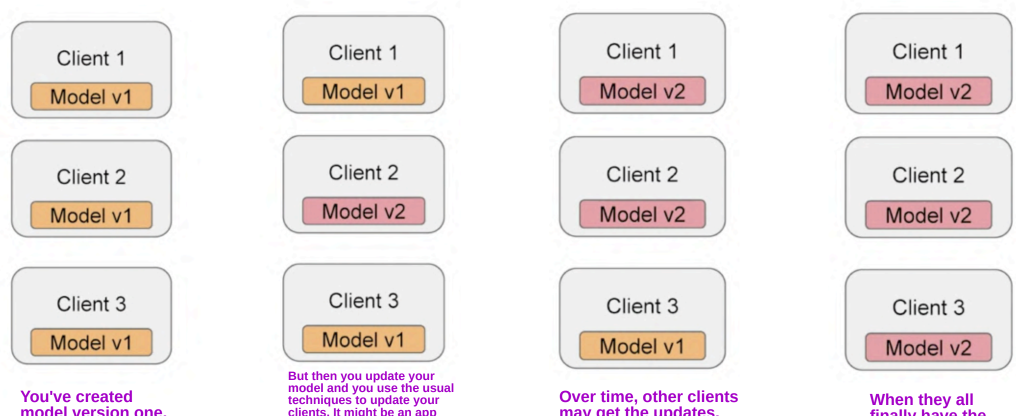


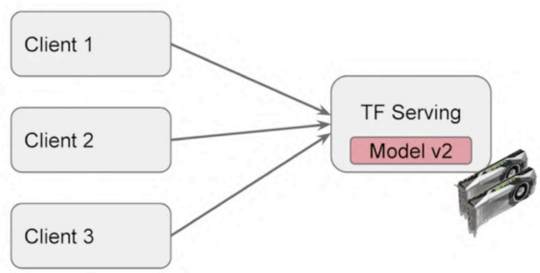
A better option, might be to have a centralized model on a server that desktops, mobile devices and more can make requests from. The server would then execute the inference for you and return the predictions back.



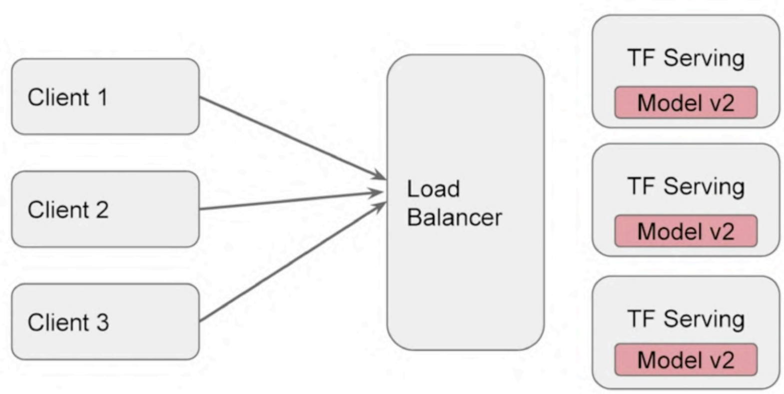
The results of which can be rendered on the device that made the call.

There is another distinct advantage of an architecture like this, and this is perhaps best illustrated with this example. Maybe you have three clients. There could be mobile devices or browsers to which you've deployed a model.



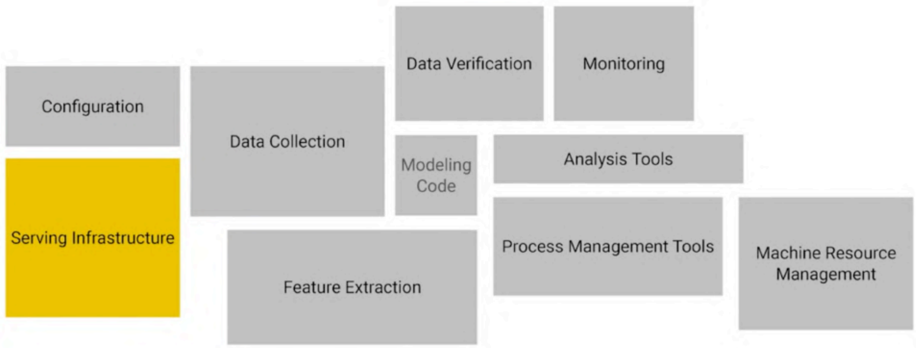


If the model isn't on the clients and instead it's centralized on a server, everybody can have the same inference experience, and that will continue even after the model is updated to a new version. Another nice thing, is that when you centralize like this, the serving environments can be upgraded based on demand with additional hardware



Of course with additional capacity based on the demand for your service by having multiple serving processes supported by some form of load balancing. This is ideal in Cloud-based environments, where you can have some form of dynamic assignment of serving processes, and you only pay for what you use.

TensorFlow Serving is part of TFX



Install TensorFlow Serving...

- Docker
- APT
- Build From Source
- PIP Packages

The first is to use Docker. This is actually the recommended way of doing it and the TensorFlow team have provided a number of Docker images that you can use. It's also recommended to do this if you want to use a GPU as everything's done for you in the Docker image.

If you want to use APT, there are packages for TensorFlow serving called 'TensorFlow Model Server'. There are 2 packages — one that is optimized using some specific compiler optimizations which should work on most machines, and a universal version that doesn't have all optimizations, but which should run almost anywhere. You'll be using this second version in this course because it installs in colab.

https://www.tensorflow.org/tfx/serving/setup

```
!echo "deb http://storage.googleapis.com/tensorflow-serving-apt stable tensorflow-model-server
tensorflow-model-server-universal"
     | tee /etc/apt/sources.list.d/tensorflow-serving.list && \

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

!apt update

!apt-get install tensorflow-model-server
```

In Colab, you can install TensorFlow serving with this code. It downloads the packages and allows you to install them into your running session.

Installation link

https://www.tensorflow.org/tfx/serving/setup

```
import tensorflow as tf
import numpy as np
from tensorflow import keras
model = tf.keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
model.fit(xs, ys, epochs=500, verbose=2)

print(model.predict([10.0]))
```

```
tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

To save the model for serving, you can then use the tf.saved_model.simple_save API. This takes the following parameters. First is your parent session.

When using Keras, the most common thing you will do is call the keras.backend.get_session.

```
tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

You can then specify the export path where you want to save the model.

```
tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

After that, you need to specify the inputs. When using TensorFlow Serving, these should be labeled input_image. And as you can see, you can access the input names by calling model.input.

```
tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

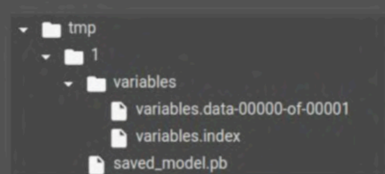
Your model outputs will also be a set of name value pairs and you can get these by iterating across all of the outputs with this code.

```
# Fetch the Keras session and save the model
# The signature definition is defined by the input and output tensors,
# and stored with the default serving key
import tempfile

MODEL_DIR = tempfile.gettempdir()
version = 1
export_path = os.path.join(MODEL_DIR, str(version))
print('export_path = {}\n'.format(export_path))
if os.path.isdir(export_path):
  print('\nAlready saved a model, cleaning up\n')
  !rm -r {export_path}

tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})

print('\nSaved model:')
!ls -l {export_path}
```

To save it to the temporary folder, this code is then used. This allows you to create a directory in the temp folder and the model will be saved there.

```
export_path = /tmp/1

Saved model:
total 44
-rw-r--r-- 1 root root 39532 Nov  3 23:35 saved_model.pb
drwxr-xr-x 2 root root  4096 Nov  3 23:35 variables
```

The result of executing this is shown here. The model is saved into the directory /tmp/1 and the files are the saved_model.pb and the variables data.




```
!saved_model_cli show --dir {export_path} --all
```

A really handy utility for understanding the structure of a model is the saved_model_cli command. You can call it like this, specifying that you want to show the model metadata giving it the directory that the model is in

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense/BiasAdd:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense/BiasAdd:0
  Method name is: tensorflow/serving/predict
```

Note the inputs. It's using the name input_image from earlier.

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense/BiasAdd:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense/BiasAdd:0
  Method name is: tensorflow/serving/predict
```

Here you can see that it's just a single value. You can ignore that leading -1.

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense/BiasAdd:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense/BiasAdd:0
  Method name is: tensorflow/serving/predict
```

Similarly, we can see the outputs. In this case, the output is called dense/BiasAdd:0

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense/BiasAdd:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense/BiasAdd:0
  Method name is: tensorflow/serving/predict
```

It's size is just a single value, just like the input.

```
os.environ["MODEL_DIR"] = MODEL_DIR

%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1
```

So now that we've saved the model, we know its location and we know the shapes of its inputs and outputs. The next thing is to get the TensorFlow model server to serve it and here's the code to achieve that.

First of all, because the model server will run as a bash command on the environment. But the variable that points to the directory containing the model is in Python, we need to tell the script where to find the model. The easiest way to do this is to write the value of the variable to an environment variable using os.environ.

```
os.environ["MODEL_DIR"] = MODEL_DIR

%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1
```

Next, we'll run a bash command and we'll ask it to execute the Tensorflow model server.

```
os.environ["MODEL_DIR"] = MODEL_DIR

%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1
```

Then we'll specify the port that we want the server to run on. In this case, it's 8501

```
os.environ["MODEL_DIR"] = MODEL_DIR

%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1
```

Tthen the name of the model. We can use whatever we want here and it will be part of the URL that's used to call the model that we'll see later.

```
os.environ["MODEL_DIR"] = MODEL_DIR

%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1
```

Then we can specify the model path and point it at the model directory.

This code sends the output to a log called server.log.

```
!tail server.log
```

You can inspect this log with the tail command and you'll see that the server is running.

```
2019-11-03 23:52:16.178660: I external/org_tensorflow/tensorflow/cc/saved_model/reader.cc:54] Reading meta graph with
tags { serve }
2019-11-03 23:52:16.179306: I external/org_tensorflow/tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU
supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2019-11-03 23:52:16.192356: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:202] Restoring SavedModel
bundle.
2019-11-03 23:52:16.198559: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:311] SavedModel load for
tags { serve }; Status: success. Took 20743 microseconds.
2019-11-03 23:52:16.198779: I tensorflow_serving/servables/tensorflow/saved_model_warmup.cc:105] No warmup data file
found at /tmp/1/assets.extra/tf_serving_warmup_requests
2019-11-03 23:52:16.198852: I tensorflow_serving/core/loader_harness.cc:87] Successfully loaded servable version
{name: helloworld version: 1}
2019-11-03 23:52:16.199907: I tensorflow_serving/model_servers/server.cc:353] Running gRPC ModelServer at
0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename not supported
2019-11-03 23:52:16.200544: I tensorflow_serving/model_servers/server.cc:373] Exporting HTTP/REST API
at:localhost:8501 ...
[evhttp_server.cc : 238] NET_LOG: Entering the event loop ...
```

It's waiting for input at Port 8501, as we specified.

Now that you have a server up and running in Colab, and it's serving the simple model that you created. Where the relationship between X and Y, where y equals 2x minus 1, has been learned.

So now we'll take a look at how to pass data to it, have it infer response from that data, and send it back to the user. What you'll see next is this, where your model is on a model serving infrastructure with TensorFlow serving. It's running in Colab at the moment, but the same code can be used to run it directly on your machine.

Clients will make a request over HTTP to your server and the server will pass that to the model, get the response. And send that back to the client, where they can then see the results of the inference.


```
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```

In order to pass your data to Serving, it needs to look like tensors. This can be achieved using JSON. So for example, this code will let us pass a list of values that I want to get inference for.

```python
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```

Because I'm running this in Colab, using Python and NumPy, I can create a NumPy list, but note the syntax. Instead of a list of values, it's a list of lists. With each list in this case, being a single value.

Later, when we look at doing inference of images, you'll notice that the images will be a list of values, and multiple images will of course be multiple lists of lists.


```python
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```


Do note, that these are also contained within a list, as you can see here.


```python
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```


Then, you'll see that we need to create a JSON blob containing a name value pair of signature name with serving default, and instances with our xs values. So where do these values come from?

```
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```

Recall earlier, when we looked at the metadata for the model, you'll notice that the serving default value comes from there. That's what the Signature-def was. You're specifying that the inputs and outputs from the model are being defined there.

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense/BiasAdd:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: dense/BiasAdd:0
  Method name is: tensorflow/serving/predict
```

```
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```

So to call the model, you'll need to tell it the signature name with this Signature-def, and the data that you want to get inferences for that are in the instances values.

```
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances": xs.tolist()})
print(data)
```

```
{"signature_name": "serving_default", "instances": [[9.0], [10.0]]}
```

Now if we print out the JSON, we'll see that it will look like this. It's two name-value pairs. The first specifying that we'll use serving default as the signature, and the second is the list of instances that will have a nine and a 10, as lists of tensors contained within a list.

```
!pip install -q requests

import requests
headers = {"content-type": "application/json"}
json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
data=data, headers=headers)

print(json_response.text)
predictions = json.loads(json_response.text)['predictions']
```

Here's the code for making a request of a server, and getting the predictions back. Let's look at this little by little.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

When making a request, we need to specify the headers, and because we're passing json to the service, we'll need to specify this. So we'll hard code the headers like this to specify.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

So with requests, we can then post a value to a URL. To do that, we have to specify the URL of the endpoint, the data that we want to post, and the requisite headers. So with requests we can then do that with this simple code.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

Note the URL structure. When we launched, we specified that it would run on port 8501, and that the models name was helloworld. So the URL structure reflects this. We're going to do a prediction so that the method at the end of the URL is called predict.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

The server will respond with json, and we can print that out with this code, and you should see a result like this.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/helloworld:predict',
                              data=data,
                              headers=headers)

print(json_response.text)

predictions = json.loads(json_response.text)['predictions']
```

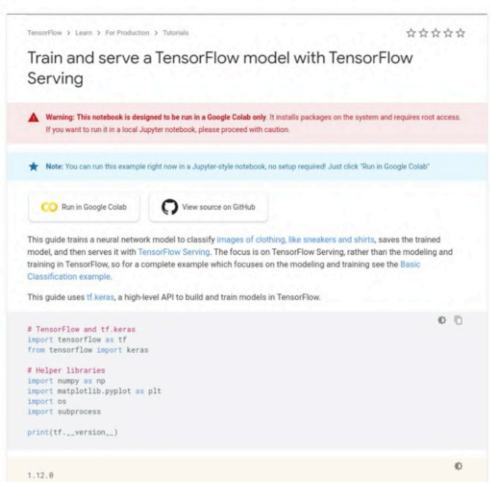
```
{
    "predictions": [[16.9865685], [18.984293]]
}
```

A list of predictions were each elements in the list is a list of results for the requisite prediction. In this case of course, each list only contains one value.

http://bit.ly/tfserving-lab1

So that's the complete run-through of getting TF serving up and running in a collab, then training a model, then serving that model before finally getting inference for that model. You can try it for yourself at this link.

https://www.tensorflow.org/tfx/serving/tutorials/Serving_REST_simple


TensorFlow > Learn > For Production > Tutorials

Train and serve a TensorFlow model with TensorFlow Serving

Warning: This notebook is designed to be run in a Google Colab only. It installs packages on the system and requires root access. If you want to run it in a local Jupyter notebook, please proceed with caution.

Note: You can run this example right now in a Jupyter-style notebook, no setup required! Just click "Run in Google Colab"

Run in Google Colab | View source on GitHub

This guide trains a neural network model to classify images of clothing, like sneakers and shirts, saves the trained model, and then serves it with TensorFlow Serving. The focus is on TensorFlow Serving, rather than the modeling and training in TensorFlow, so for a complete example which focuses on the modeling and training see the Basic Classification example.

This guide uses tf.keras, a high-level API to build and train models in TensorFlow.

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
import os
import subprocess

print(tf.__version__)
```

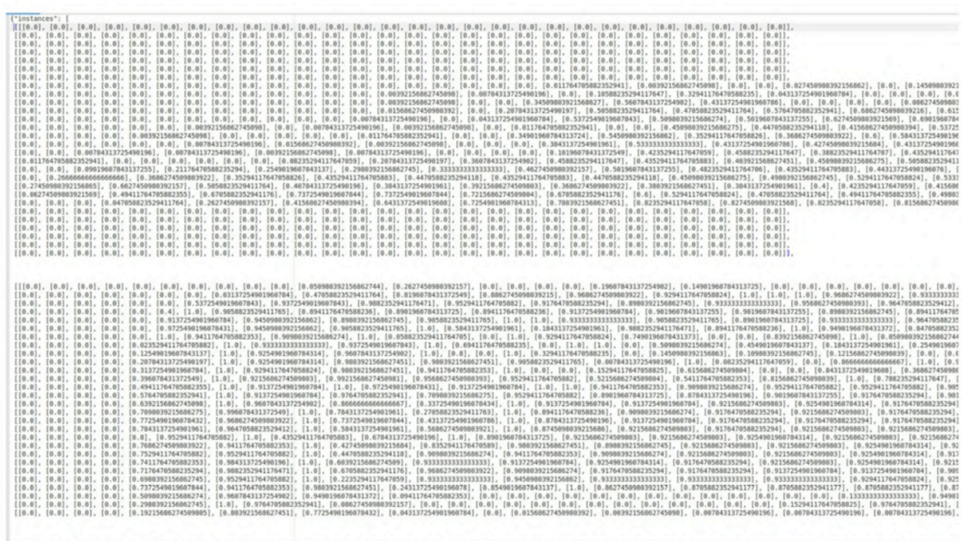
```
1.12.0
```

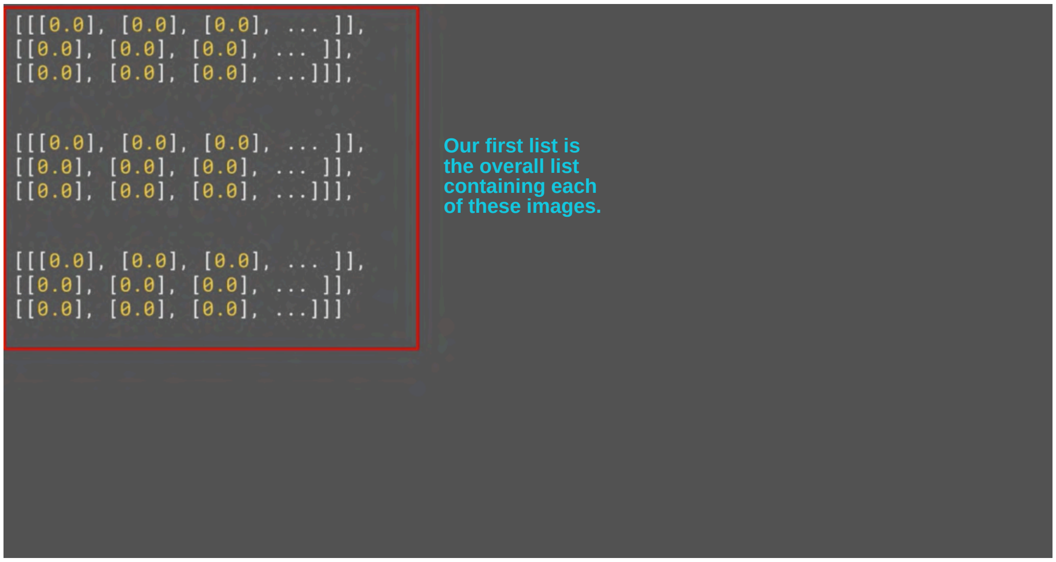
```
import json

data = json.dumps({"signature_name": "serving_default",
                   "instances": test_images[0:3].tolist()})

print('Data: {} ... {}'.format(data[:50], data[len(data)-52:]))

print(data)
```

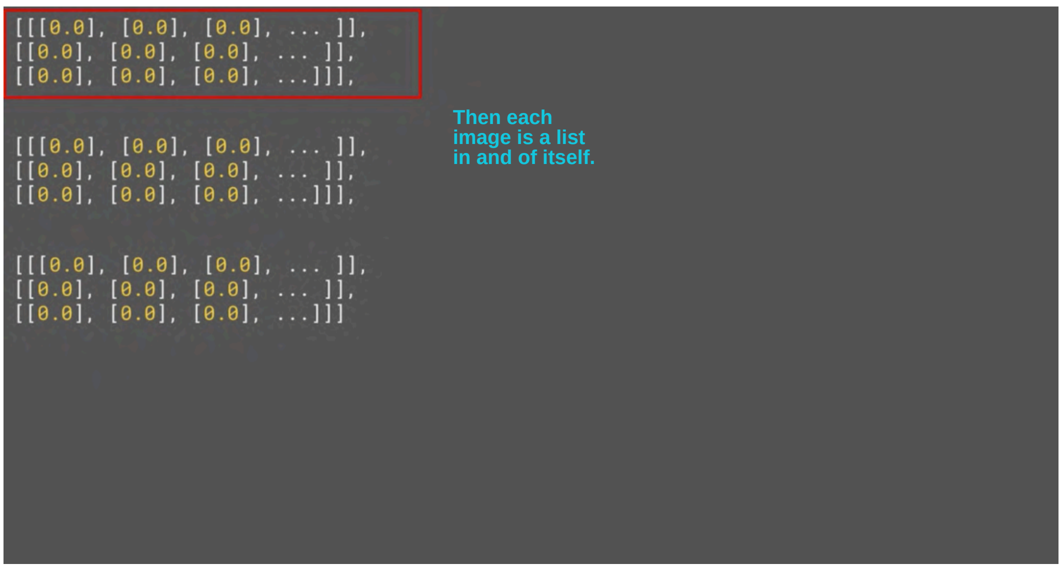
In this case, the images for fashion MNIST are stored in a raise of 28 by 28 values. And normalize gray scale with 0 being black and 1 being white, and everything else being the shades in between.

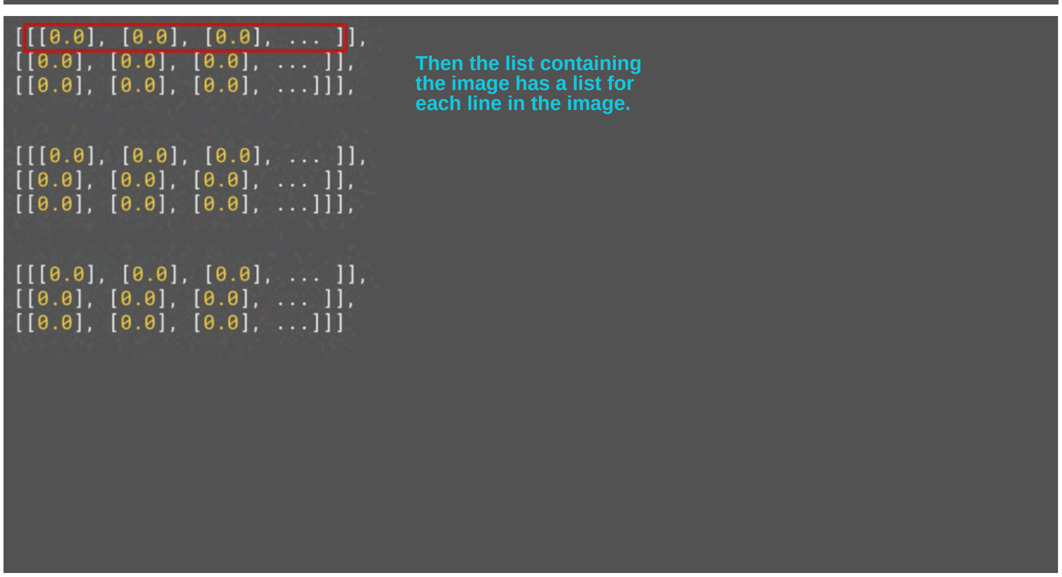
So if you consider each image to be a 28 by 28 value, it's represented as a list of 28 items, each of which contains another 28 items.

So you can consider an image to be a list of lists. And if you have multiple images, then you'll have a list of these namely a list of lists of lists. So if we are to print out the data as we can see here, we'll get something that looks like this.




[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]]
Our first list is the overall list containing each of these images.


[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]]
Then each image is a list in and of itself.


[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]],
[[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]],
[[0.0], [0.0], [0.0], ...]]]
Then the list containing the image has a list for each line in the image.

```
!pip install -q requests

import requests

headers = {"content-type": "application/json"}

json_response = requests.post('http://localhost:8501/v1/models/fashion_model:predict',
                              data=data, headers=headers)

predictions = json.loads(json_response.text)['predictions']
```

Now, if you want to call the end point, you can use exactly the same code as before. Our data this time contains the list of images as we just previously discussed.

```
[
[5.77123615e-07, 2.66907847e-08, 4.7217938e-08, 1.97792871e-09, 5.31984341e-08,
0.00734644197, 3.1462946e-07, 0.0439051725, 0.000500570168, 0.948246837],

[0.00227244, 6.12080342e-09, 0.967876315, 3.0579281e-06, 0.0183339939, 3.18483538e-11,
0.011510049, 1.38639566e-14, 4.19033222e-06, 4.40264526e-11],

[1.45221502e-05, 0.999841571, 3.96758715e-08, 0.000131023204, 1.22008023e-05,
1.18227668e-08, 5.97860179e-08, 1.31281848e-08, 5.49047854e-07, 2.97885189e-10]
]
```

The results will be a list of predictions coming back and each of these predictions is a list of the probabilities for a particular class. Fashion MNIST has 10 classes.

```
[
[5.77123615e-07, 2.66907847e-08, 4.7217938e-08, 1.97792871e-09, 5.31984341e-08,
0.00734644197, 3.1462946e-07, 0.0439051725, 0.000500570168, 0.948246837],

[0.00227244, 6.12080342e-09, 0.967876315, 3.0579281e-06, 0.0183339939, 3.18483538e-11,
0.011510049, 1.38639566e-14, 4.19033222e-06, 4.40264526e-11],

[1.45221502e-05, 0.999841571, 3.96758715e-08, 0.000131023204, 1.22008023e-05,
1.18227668e-08, 5.97860179e-08, 1.31281848e-08, 5.49047854e-07, 2.97885189e-10]
]
```

So if you inspect the data, you can see that some of the classes ended up with a very high confidence of being in the respective class.

```
show(0,
'The model saw {} (class {}), and it was actually a {} (class {})'.format(
      class_names[np.argmax(predictions[0])], test_labels[0],
      class_names[np.argmax(predictions[0])], test_labels[0]))
```

You can also pick the top value using ARG Max. So this code will output the image and the prediction for it so that you can see if it's correct.

The model thought this was a Ankle boot (class 9), and it was actually a Ankle boot (class 9)