Copyright Notice

Model Serving

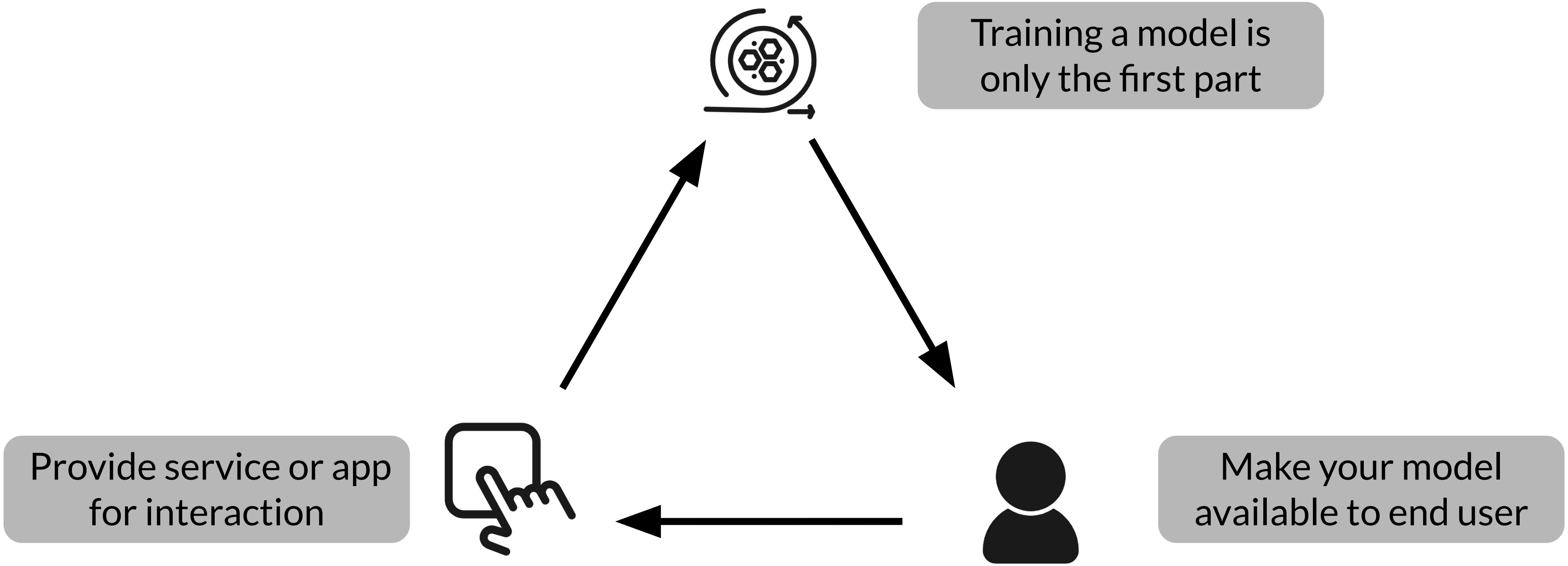
Welcome

Introduction to Model Serving

DeepLearning.AI

Introduction

What exactly is Serving a Model?



Model Serving Patterns

- A model,
- An interpreter, and
- Input data

Inference

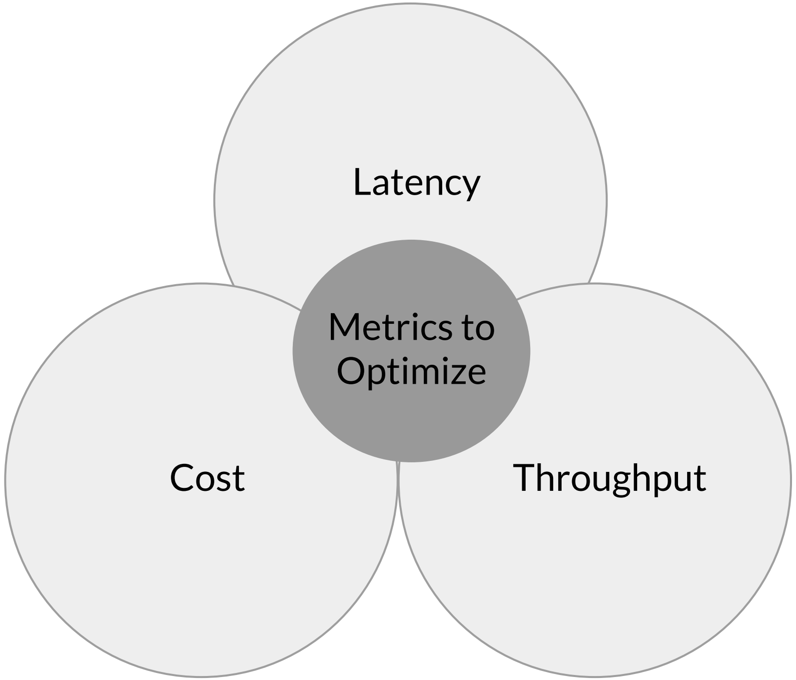
ML workflows

- Model training
- Model prediction

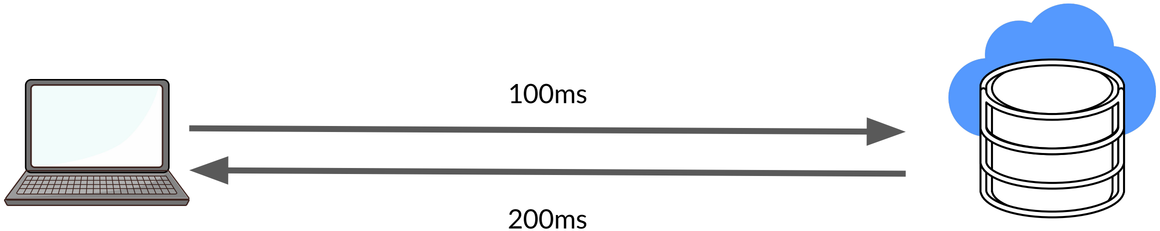
Batch inference

Realtime inference

Important Metrics



Latency



Latency = 100 + 200 = 300ms

- Delay between user's action and response of application to user's action.
- Latency of the whole process, starting from sending data to server, performing inference using model and returning response.
- Minimal latency is a key requirement to maintain customer satisfaction.

Throughput

- ➔ Throughput -> Number of successful requests served per unit time say one second.
- ➔ In some applications only throughput is important and not latency.

Cost

- The cost associated with each inference should be minimised.
  - Important Infrastructure requirements that are expensive:
    - CPU
    - Hardware Accelerators like GPU
    - Caching infrastructure for faster data retrieval.

Minimizing Latency, Maximizing Throughput

Minimizing Latency

- Airline Recommendation Service
- Reduce latency for user satisfaction

Maximizing Throughput

- Airline recommendation service faces high load of inference requests per second.

Scale infrastructure (number of servers, caching requirements etc.) to meet requirements.

Balance Cost, Latency and Throughput

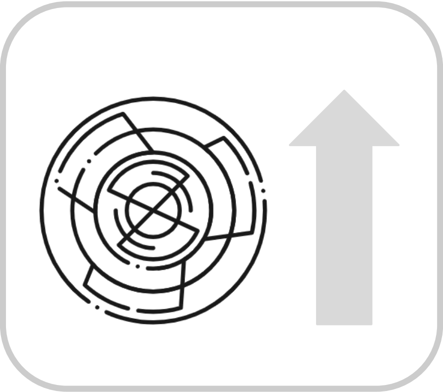
- Cost increases as infrastructure is scaled
- In applications where latency and throughput can suffer slightly:
  - Reduce costs by GPU sharing
  - Multi-model serving etc.,
  - Optimizing models used for inference

DeepLearning.AI

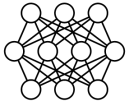
Introduction

Resources and Requirements for Serving Models

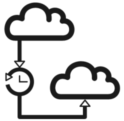
Optimizing Models for Serving



Model Complexity



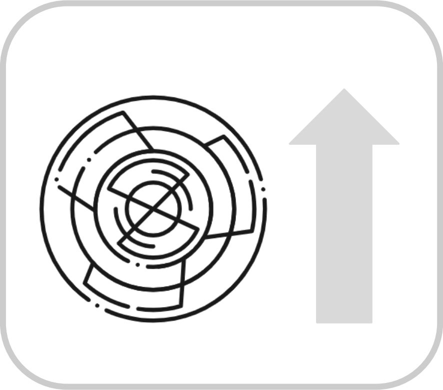
Model Size
Complex functions



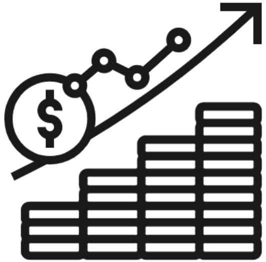
Prediction Latency

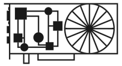
Prediction Accuracy

As Model Complexity Increases Cost Increases

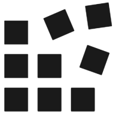


Model Complexity





GPU
TPU

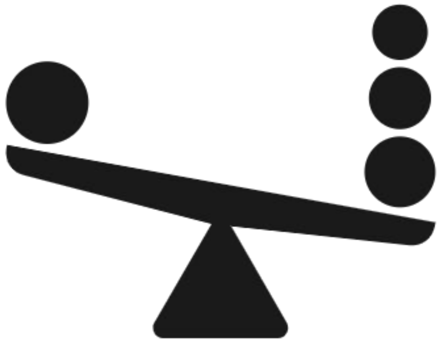


Model
registry

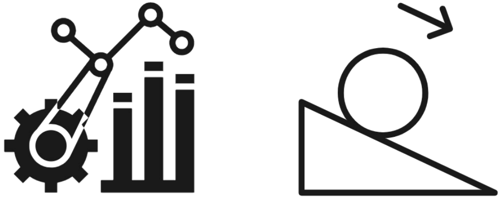
Maintenance
burden

Balancing Cost and Complexity

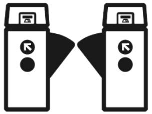
The challenge for ML practitioners is to balance complexity and cost.



Optimizing and Satisficing Metrics

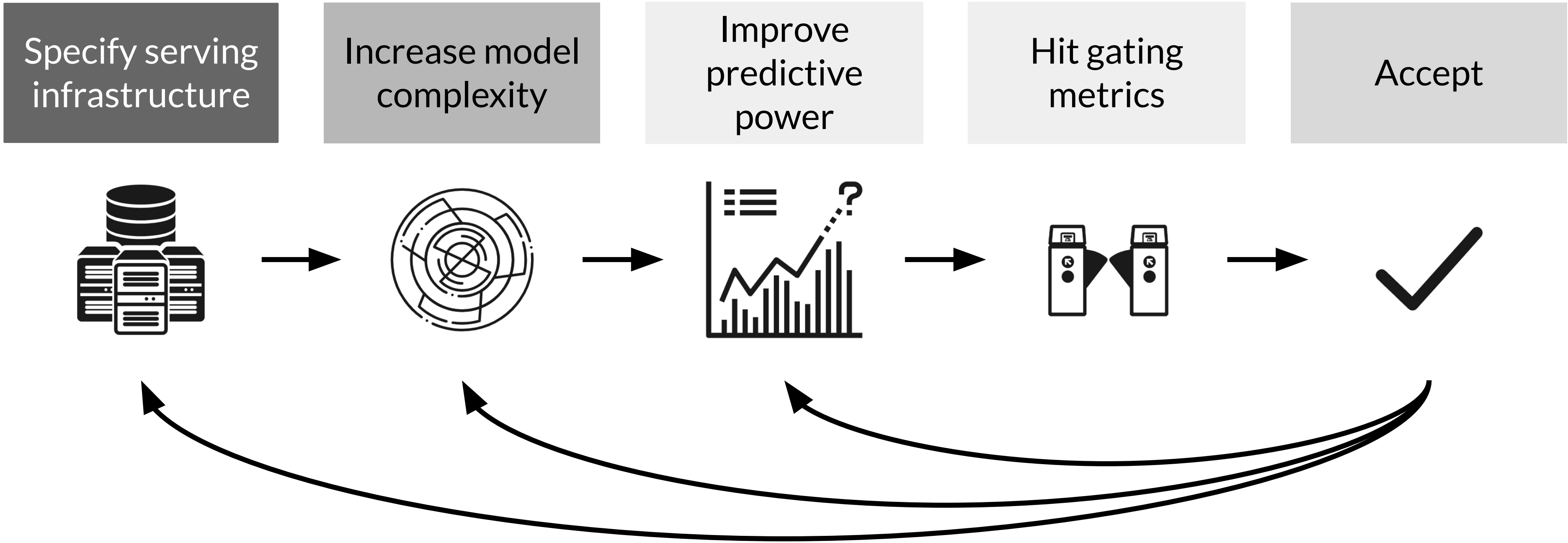


Model's optimizing metric:
- Accuracy
- Precision
- Recall

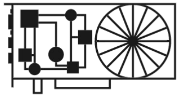


Satisficing (Gating) metric:
- Latency
- Model Size
- GPU load

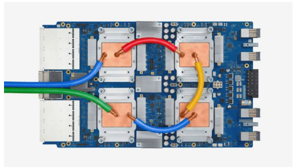
Optimizing and Satisficing Metrics



Use of Accelerators in Serving Infrastructure

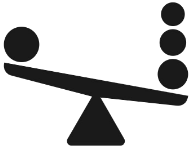


GPUs for parallel throughput

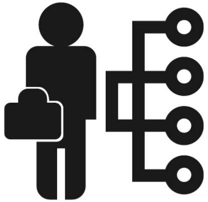


TPUs for complex models and large batches

Hardware choices impact cost



Balancing complexity and hardware choices

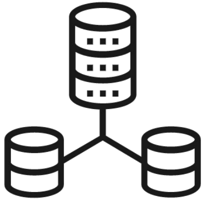


Choices made at organizational level

Maintaining Input Feature Lookup

- Prediction request to your ML model might not provide all features required for prediction
- For example, estimating how long food delivery will require accessing features from a data store:
  - Incoming orders (not included in request)
  - Outstanding orders per minute in the past hour
- Additional pre-computed or aggregated features might be read in real-time from a data store
- Providing that data store is a cost

NoSQL Databases: Caching and Feature Lookup



NoSQL Databases

Google Cloud Memorystore
In memory cache, sub-millisecond read latency

Google Cloud Firestore
Scaleable, can handle slowly changing data, millisecond read latency

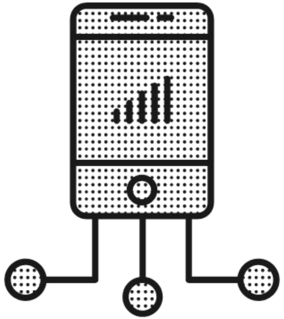
Google Cloud Bigtable
Scaleable, handles dynamically changing data, millisecond read latency

Amazon DynamoDB
Single digit millisecond read latency, in memory cache available

Expensive. Carefully choose caching requirements

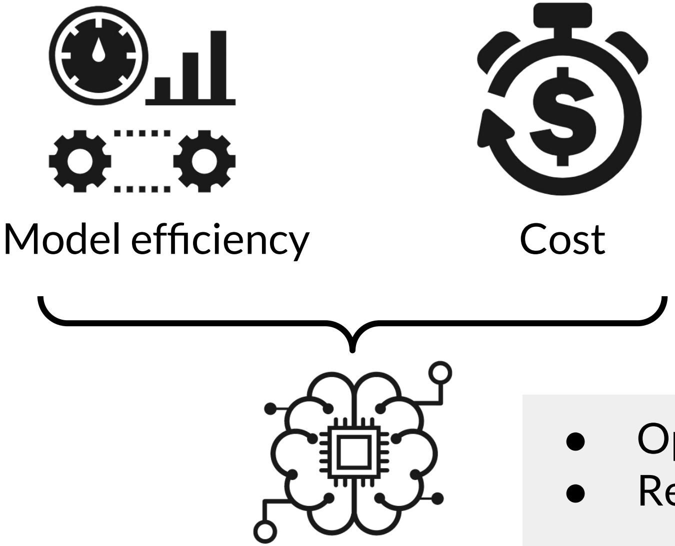
Model Deployments

- Huge data centers



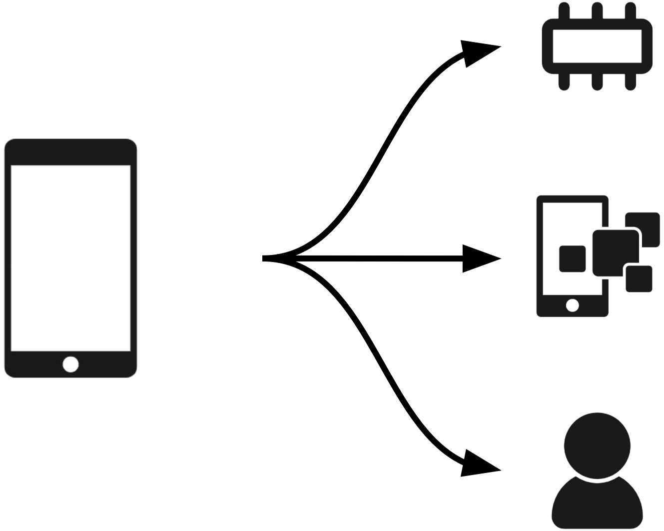
- Embedded devices

Running in Huge Data Centers



- Optimize resource utilization
- Reduce cost

Constrained Environment: Mobile Phone

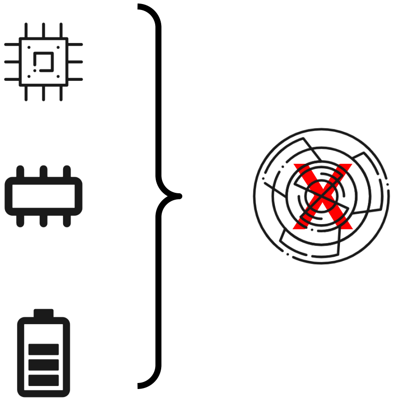


Average GPU memory size < 4GB.

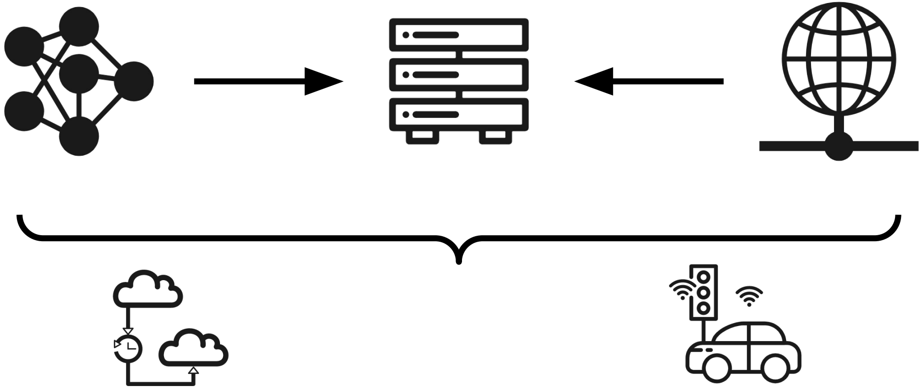
Average Android app size ≅ 11MB

Users might not install your app if it uses too much storage

Restrictions in a Constrained Environment



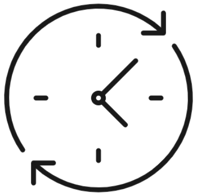
Large, complex models cannot be deployed to edge devices



Will not work when prediction latency is important. E.g. autonomous car.

Prediction Latency is Almost Always Important

- Opt for on-device inference whenever possible
  - Enhances user experience by reducing the response time of your app



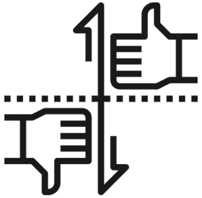
Millisecond turnaround

Model efficiency

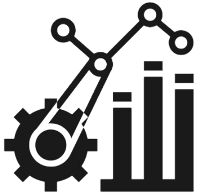
Cost

Choose Best Model for the Task

Other Strategies

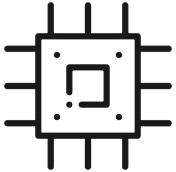


Profile and Benchmark



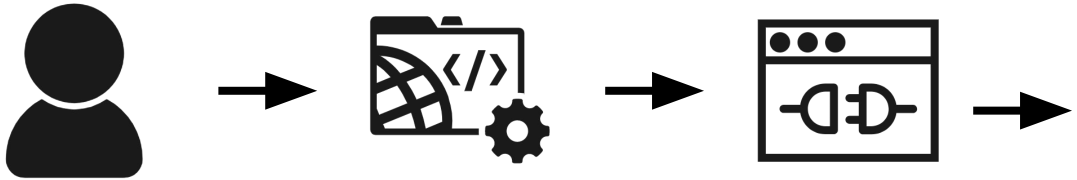
Optimize Operators

Optimize Model



Tweak Threads

Web Applications for Users

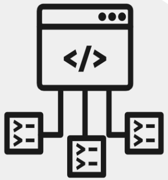


Users make requests via web application

Model wrapped as API service

Python:

- FastAPI
- Flask
- Django



Java:

- Spring
- Apache Tomcat

Serving systems for easy deployment

- Centralized model deployment
- Predictions as service

Eliminates need for custom web applications

Deployment just a few lines of code away

Easy to rollback/update models on the fly

Clipper

- Open-source project from UC Berkeley
- Multiple modeling frameworks
- RESTful API
- Cluster and resources management
- Settings for reliable latency

TensorFlow Serving

Open-source project from Google

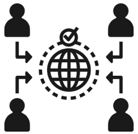
Serve TensorFlow models easily

Extensible to serve other model types

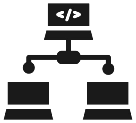
Uses REST and gRPC protocol

Version manager

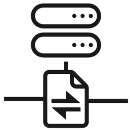
Advantages of Serving with a Managed Service



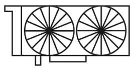
- Realtime endpoint for low-latency predictions on massive batches



- Deployment of models trained on premises or on the Google Cloud Platform



- Scale automatically based on traffic



- Use GPU/TPU for faster predictions

DeepLearning.AI

TensorFlow Serving

Installing and Running TensorFlow Serving

Install TensorFlow Serving

- Docker Images:
    - Easiest and most recommended method
    - Easiest way to get GPU support with TF Serving

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

Install TensorFlow Serving

Available Binaries	
tensorflow-model-server	**tensorflow-model-server-universal**:
1. Fully optimized server 2. Uses some platform specific compiler optimizations 3. May not work on older machines	1. Compiled with basic optimizations 2. Doesn't include platform specific instruction sets 3. Works on most of the machines

Install TensorFlow Serving

- Building From Source
  - See the complete documentation

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

- Install using Aptitude (apt-get) on a Debian-based Linux system

Install TensorFlow Serving

```
!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
```

Import the MNIST Dataset

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
# Scale the values of the arrays below to be between 0.0 and 1.0.
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Import the MNIST Dataset

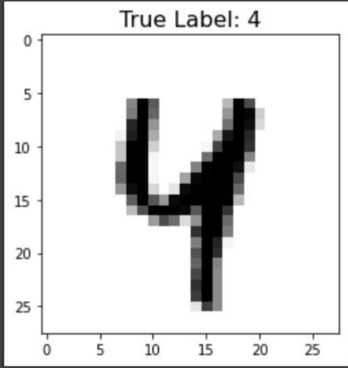
```
# Reshape the arrays below.
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)
print('\ntrain_images.shape: {}, of {}'.format(train_images.shape,
train_images.dtype))
print('test_images.shape: {}, of {}'.format(test_images.shape, test_images.dtype))

train_images.shape: (60000, 28, 28, 1), of float64
test_images.shape: (10000, 28, 28, 1), of float64
```

Look at a Sample Image

```
idx = 42

plt.imshow(test_images[idx].reshape(28,28), cmap=plt.cm.binary)
plt.title('True Label: {}'.format(test_labels[idx]), fontdict={'size': 16})
plt.show()
```



Build a Model

```
# Create a model.
model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(input_shape=(28,28,1), filters=8, kernel_size=3,
                               strides=2, activation='relu', name='Conv1'),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax, name='Softmax')
])

model.summary()
```

Train the Model

```
# Configure the model for training.
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

epochs = 5


# Train the model.
history = model.fit(train_images, train_labels, epochs=epochs)
```

Evaluate the Model

```
# Evaluate the model on the test images.
results_eval = model.evaluate(test_images, test_labels, verbose=0)

for metric, value in zip(model.metrics_names, results_eval):
    print(metric + ': {:.3}'.format(value))

loss: 0.098
accuracy: 0.969
```

Save the Model

```
MODEL_DIR = tempfile.gettempdir()
version = 1
export_path = os.path.join(MODEL_DIR, str(version))

if os.path.isdir(export_path):
    print('\n Already saved a model, cleaning up\n')
    !rm -r {export_path}

model.save(export_path, save_format="tf")

print('\nexport_path = {}'.format(export_path))
!ls -l {export_path}
```

Launch Your Saved Model

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

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

Send an Inference Request

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

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

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

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

Plot Predictions

```
plt.figure(figsize=(10,15))

for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(test_images[i].reshape(28,28), cmap = plt.cm.binary)
    plt.axis('off')
    color = 'green' if np.argmax(predictions[i]) == test_labels[i] else 'red'
    plt.title('Prediction: {}\n True Label: {}'.format(np.argmax(predictions[i]),
test_labels[i]), color=color)

plt.show()
```

Results Demo