Deploying Machine Learning Models in Production

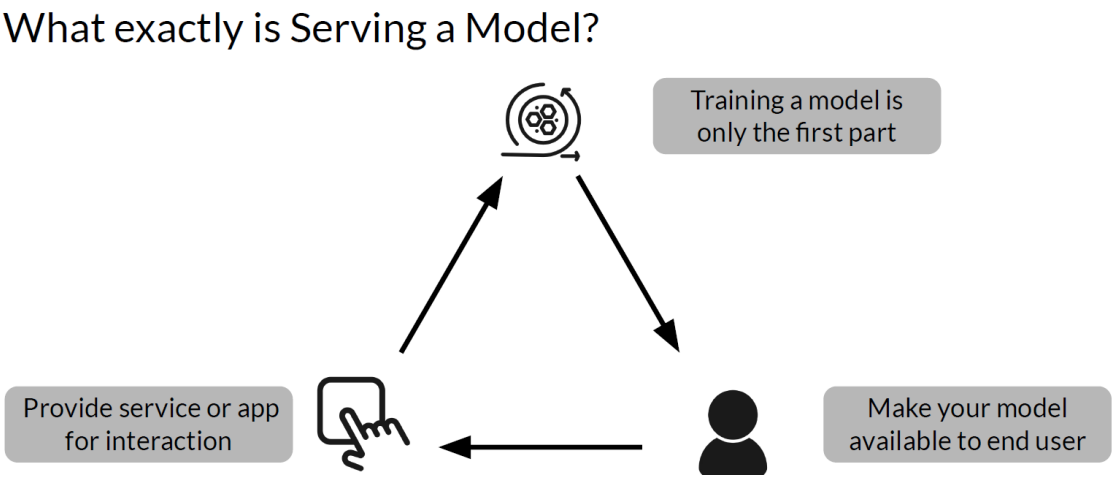
In the fourth course of Machine Learning Engineering for Production Specialization, you will learn how to deploy ML models and make them available to end-users. You will build scalable and reliable hardware infrastructure to deliver inference requests both in real-time and batch depending on the use case. You will also implement workflow automation and progressive delivery that complies with current MLOps practices to keep your production system running. Additionally, you will continuously monitor your system to detect model decay, remediate performance drops, and avoid system failures so it can continuously operate at all times.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

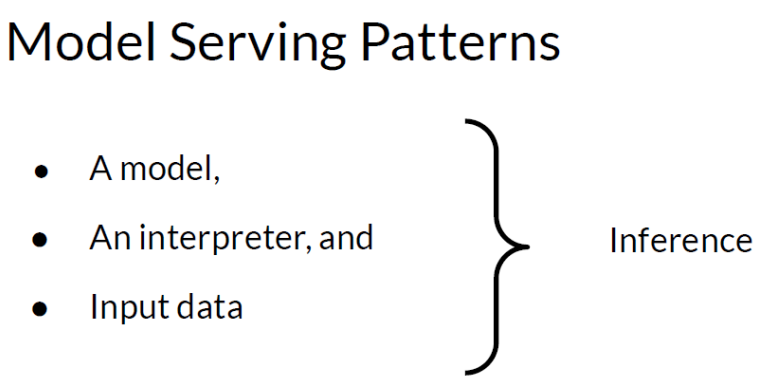
Week 1: Model Serving Introduction

Contents

Introduction to Model Serving



- Hi, everybody. Welcome to this course where we'll discuss model serving patterns and infrastructures. We'll also look at management and delivery as well as monitoring.
- This week, I'll give you an introduction to model serving, where you will explore methods for deploying models before getting hands-on and trying it for yourself.
- Don't worry, you're not going to need a serving infrastructure of your own, you're going to be able to do all of the exercises using Colab. Let's get started.
- First, we'll discuss model performance and resource requirements, starting with the discussion of batch inference.
- What exactly do we mean by serving a model? Well, that's a great question.
- Training a good machine learning model is only the first part.
- You do need to make your model available to your end-users, and you do this by either providing access to the model on your server, which we'll explore this week.
- Serving also includes the facility to deploy a model file to an application, and this is a scenario that's usually found with mobile ML.



- When serving an ML model in a production environment, you should consider three key components.
- There's the model itself, there's an interpreter for the execution, and there's input data.
- These components are used by an inference process which is aimed at getting the data to be ingested by a model in order to compute predictions.

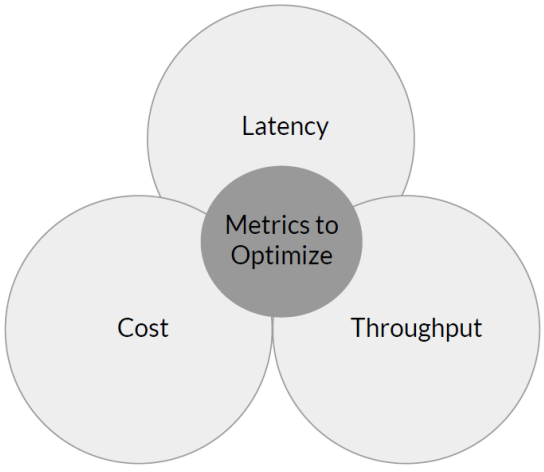
ML workflows

- Model training
- Model prediction

} Batch inference, Realtime inference

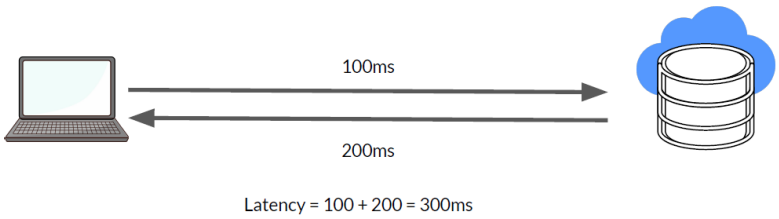
- A high level look at ML workflows boils down to something like this; we take into consideration model training and prediction while understanding the types of inference or prediction that you want to do with your model.
- Model **training** is performed either with **offline**, also known as **batch** or static learning, where the model is trained on a set of already collected data.
- After deploying to the production environment, the ML model remains constant until it's retrained because the model will see a lot of real life data and then it becomes stale quite quickly.
- This phenomenon is called **model decay**, and it's something that should be carefully monitored.
- There's also **online** learning, also known sometimes as **dynamic** learning.
- Here the model is regularly being retrained as new data arrives, for example, as data **streams**.
- This is usually the case for ML systems that use time series data, such as sensors or stock or anything like that, and the idea is to accommodate the temporal effects in the ML model.
- Then **when using your model for prediction, there's also two main types.**
- With batch predictions, the deployed ML model makes a set of predictions based on historical input data.
- This is often sufficient for data that's not time dependent or when it's not critical to obtain real-time predictions as output.
- Or with real-time predictions, also known as on-demand predictions, these predictions are generated in real time using the input data that's available at the time of the request.

Important Metrics



- When we talk about optimizing **online inference**, most of the material surrounding it will first give us an overview about the important metrics to optimize, namely: **latency, throughput, and cost**.

Latency



- 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.

- Latency is the **delay** between a **user's action** and the **response** of the **application** to the user's action.
- In the case of ML inference, it's the whole process of online inference, starting from sending data to the server, performing inference using the model, and then returning the response.
- Most applications are user-facing, so minimal latency is a key requirement to maintaining your customer satisfaction.
- For example, your users might complain the app that suggests hotels is too slow to refresh the search results based on a user's input.

Throughput

➔ Throughput -> Number of successful requests served per unit time say one second.

➔ In some applications only throughput is important and not latency.

- Throughput is the number of successful requests served in a unit of time.
- An example here could be if you want your models to process large amounts of data, for example, video from security cameras at a very high fidelity in order to assure that any security events are spotted quickly.

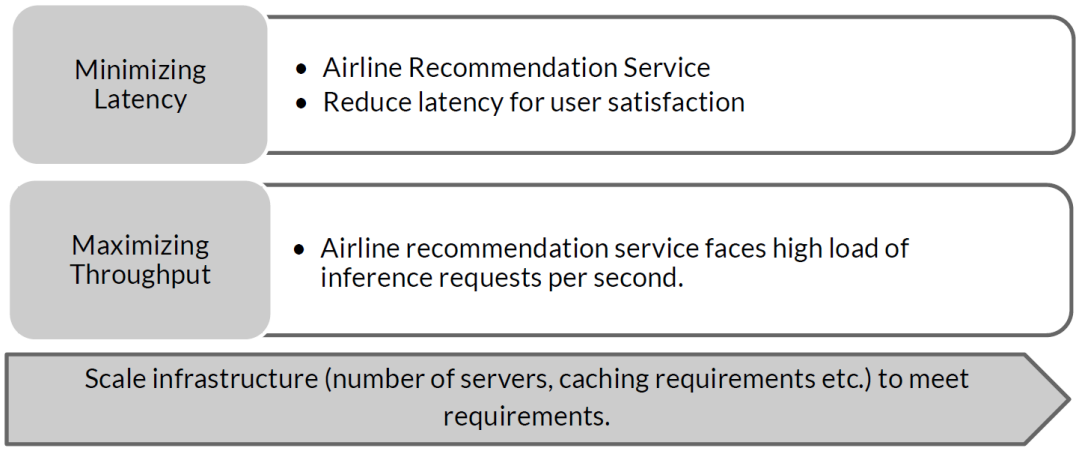
Cost

- 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.

- We should always try to remember and factor in the cost associated with each inference.
- Your serving infrastructure will always have associated costs.

- You should consider the cost for things like CPUs, hardware accelerators like GPUs, and caches for storing input features for easy retrieval with minimum latency.

Minimizing Latency, Maximizing Throughput



- Many customer facing applications have the aim to minimize this latency while also maximizing throughput.
- An example of this could be an airline recommendations websites.
- This gives you recommendations about the best flights that are available to you based on your inputs.
- Smarter systems could be running ML models to predict the best options for their users.
- At some points in time, such as holidays, this application could face a very high load of users.
- You have the burden of providing low latency, giving your users results quickly with high-throughput, having a model serving infrastructure that can handle that high load.
- We can scale the infrastructure used to meet these thresholds of response time and throughput, but this can increase the cost proportionally.

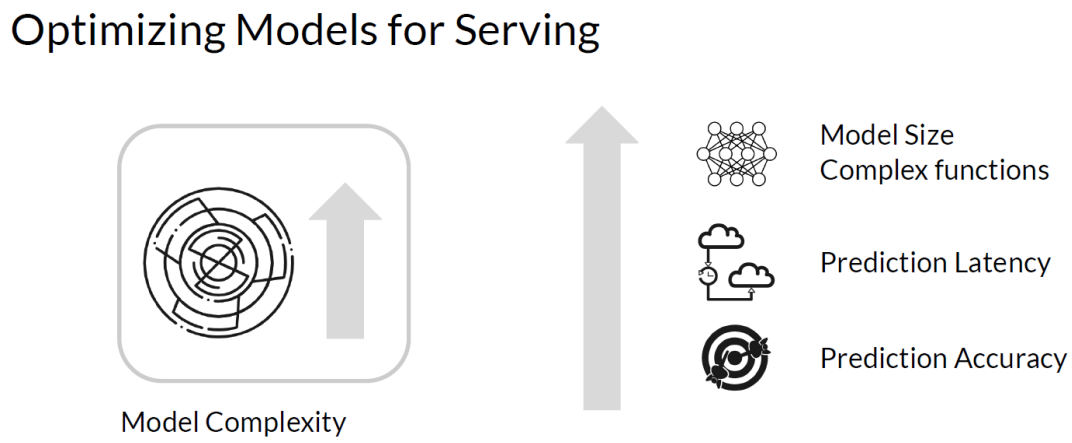
Balance Cost, Latency and Throughput

- 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

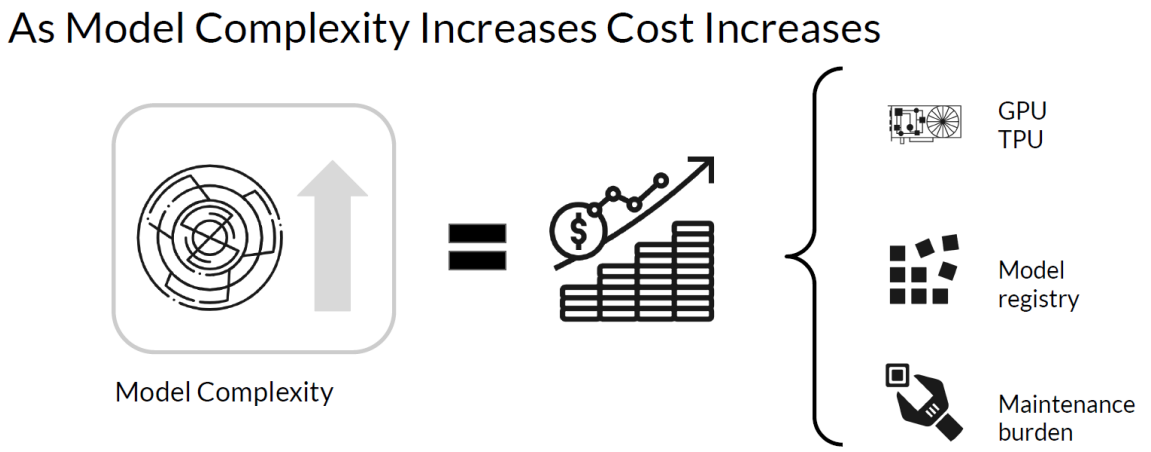
- The cost of running your ML model in production can increase sharply as your infrastructure is scaled to handle latency and throughput.
- This leads you to play a **balancing** game between costs and customer satisfaction.
- There's no perfect answer, and there are always tradeoffs.
- But fortunately, there are tactics you can use to try to minimize any impact on your customer while you attempt to control cost.
- These may include reducing costs by sharing assets like GPU's, using multiple models to increase throughput, and perhaps even exploring optimizing your models.

- Now that you've seen the basic concepts that you need to understand for hosting models on a server and understanding the decisions that you may need to make, we'll next look at some of the tactics that will help you in these decisions, including maybe even not using a server and exploring the deployment of a model directly into a user's hands on their mobile or edge devices.

Introduction to Model Serving Infrastructure



- We saw in the previous lesson that when deploying a model to production environment, we might have some decisions to make in order to maximize throughput while minimizing latencies and cost.
- Let's now look at some of the issues with **resource costs and some constraints** that we might encounter.
- There are good reasons why models often become complex
- In an effort to find ways to increase accuracy or model more complex relationships, there is a natural impulse to imply more complex model architectures, including more and more features.
- And this often results in longer prediction latencies and but hopefully a boost in prediction accuracy.



- However, as models become more complex and more and more features are included, the resource requirements increase for every part of the training and serving infrastructure
- Increased resource requirements means increased cost, and increased hardware requirements management of larger model registries, and this results in a higher support and maintenance burden.

Balancing Cost and Complexity

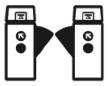
The challenge for ML practitioners is to balance complexity and cost.

- As in many things in life, the key is to find the right balance
- Finding that right balance between cost and complexity is a skill that seasoned practitioners will build up over time.
- So there's a tradeoff between the model's predictive effectiveness and the speed of its prediction latency.

Optimizing and Satisficing Metrics

Model's optimizing metric:

- Accuracy
- Precision
- Recall

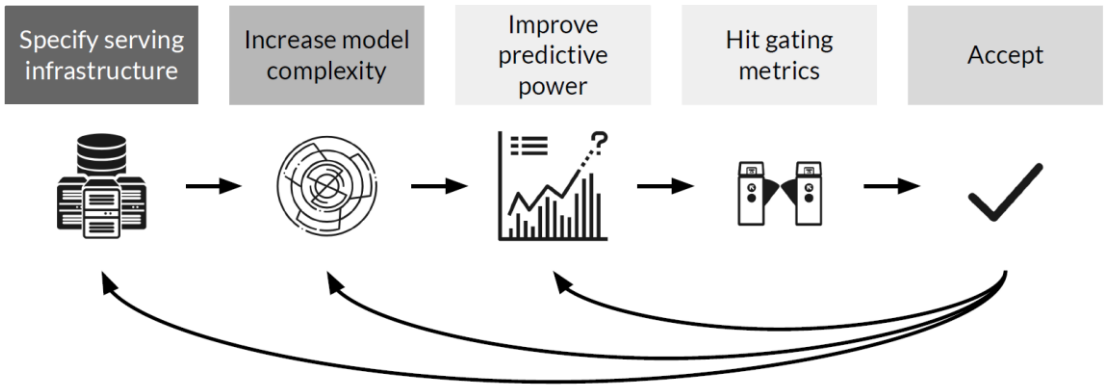


Satisficing (Gating) metric:

- Latency
- Model Size
- GPU load

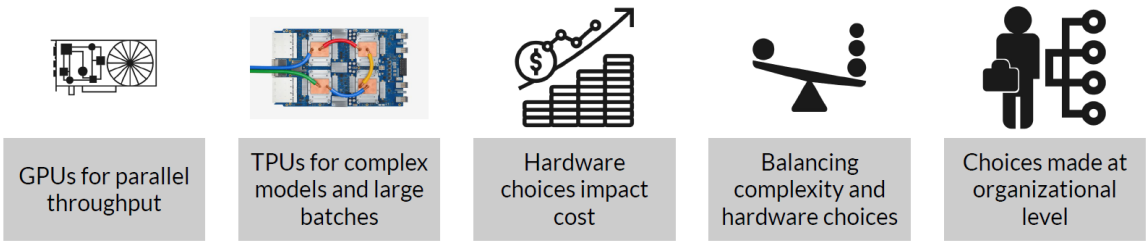
- Depending on the use case, you need to decide on **two** metrics.
- There's the model's **optimizing metric** which reflects the model's predictive effectiveness and this includes things like accuracy, precision, recall, and so on.
- Good values in these metrics is a strong signal about the quality of your model.
- And then there's the model's **gating (or satisficing) metric**, and this reflects an **operational constraint** that the model has to satisfy, such as prediction latency.
- Optimizing metrics measure the model's predictive effectiveness while satisficing metrics specify operational constraints.
- So for example, you might set a latency threshold to a particular value, such as 200 milliseconds, and any model that doesn't meet this threshold is not going to be accepted.
- Another example of a gating metric is the size of the model.
- If you plan on deploying a model too low spec hardware like mobile and embedded devices, this is of course very important.

Optimizing and Satisficing Metrics



- One approach you can take is to specify the serving infrastructure, CPU, GPU and all that.
- And then start increasing your model complexity to improve your model's predictive power until you hit one or more of your gating metrics on that infrastructure.
- Then you can assess the results and either accept the model as it is, or work to improve accuracy, or reduce complexity or make the decision to increase the specifications of the serving infrastructure.

Use of Accelerators in Serving Infrastructure



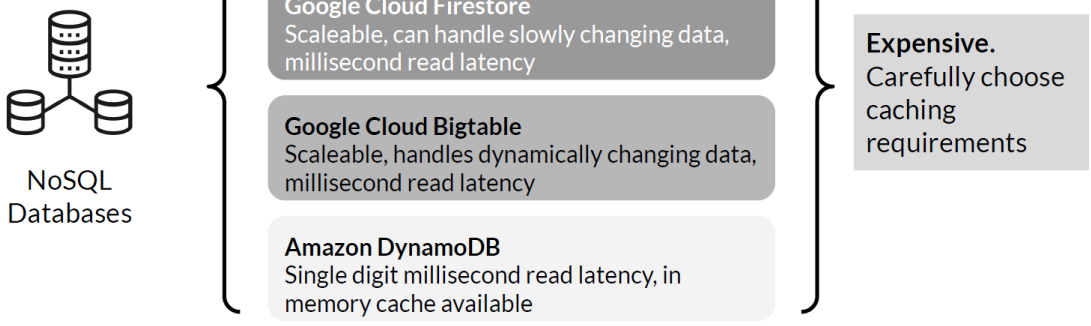
- One of the factors to consider when designing your server and training infrastructure is the use of accelerators such as GPUs and TPUs.
- Now, each of these have different advantages but they also have costs and potentially limitations.
- **GPUs tend to be optimized for parallel throughput** and they are often used in training infrastructure, while **TPUs**, as well as being useful in training, have **advantages for large complex models and large batch sizes**, **especially during inference**.
- These decisions can have a significant effect on your projects budget.
- So there's also a tradeoff between applying a large number of less powerful accelerators and using a smaller number of more powerful accelerators.
- Often when working with a team or department, these choices will need to be made for a broad range of models, and not just for the new model that you're working on at this moment because there are always going to be shared resources.

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

- The prediction request to your ML model might not provide all of the features required for prediction.
- Some of the features may also need to be **pre computed or aggregated and then read in real time** from a data store.
- Take for example of food delivery app that needs to predict the estimated time for an order delivery.
- This is based on a number of features like **current traffic conditions**.
- There are also some that can be read from a data store like the list of incoming orders, the number of outstanding orders per minute in the last hour, and stuff like that.
- You'll need **powerful caches to retrieve this data with low latency since the delivery time has to be updated in real time**.
- You cannot wait many seconds for retrieving data from the database.
- And of course this has cost implications.

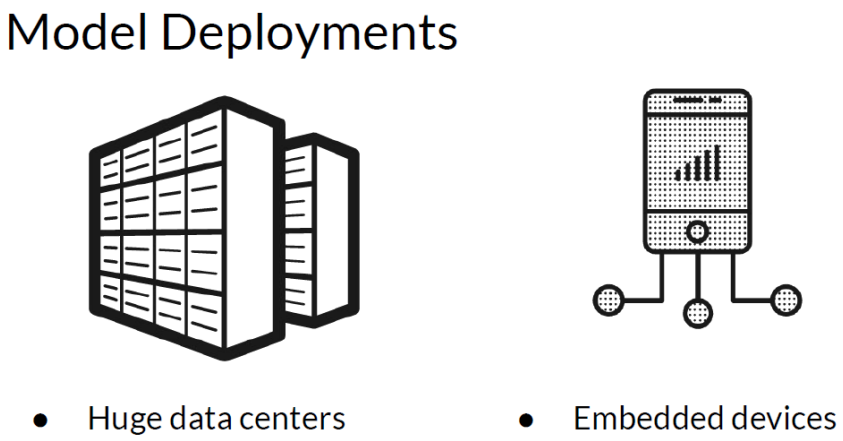
NoSQL Databases: Caching and Feature Lookup



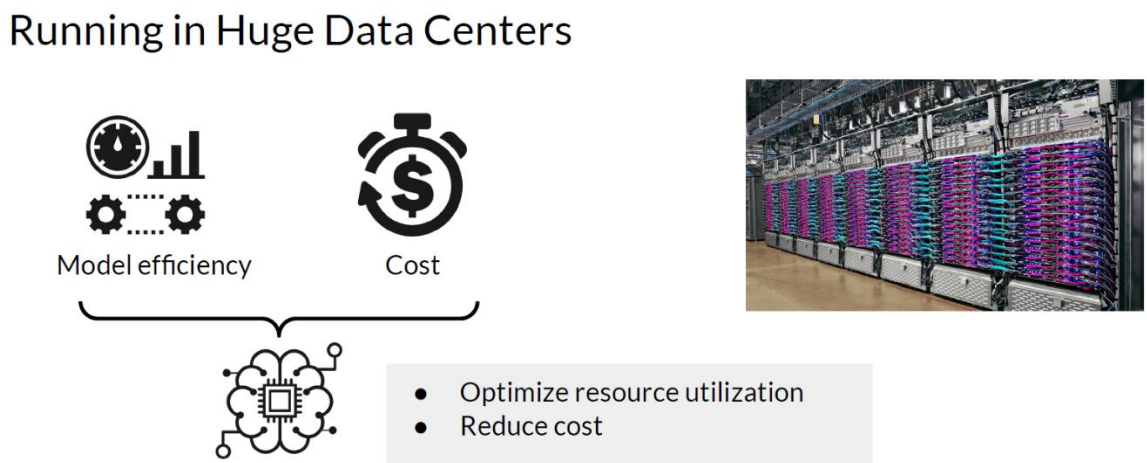
- **NoSQL or NoSQL databases are a good solution to implement caching and feature look up**.
- And there are various options available. If you need sub-millisecond **read latency** on a limited amount of quickly changing data retrieved by a few 1000 clients, one good choice is **Google Cloud Memorystore**, which is a fully managed version of latency in memory cache.
- And of course there were also very good open source options.

- If you need millisecond read latency on slowly changing data where the storage scales automatically, one good choices **Google Cloud Firestore**.
- If you need millisecond read latency on dynamically changing data using a store that can scale linearly with heavy reads and writes, one good choice of course is **Google Cloud Bigtable**.
- **Amazon's DynamoDB** is also a good choice for scalable low read latency database with an in memory cache.
- Adding caches speeds up feature look up while reducing prediction retrieval latency.
- You have to carefully choose from the different available offerings based on your requirements and then balance that with your budget constraints.

Deployment Options



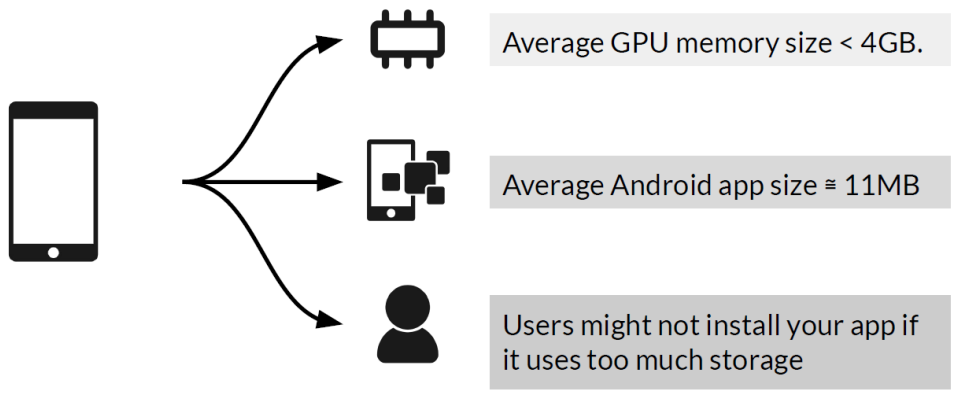
- Returning to the question of where should I deploy my model?
- There are primarily two choices.
- You can have a **centralized model in a data center that's accessed via a remote call**.
- Or you can **distribute instances of your model to your users** so they can use it locally such as a mobile or embedded system.



- In our data centers, costs and efficiency are important at any scale, even when you have large resources in huge data center.

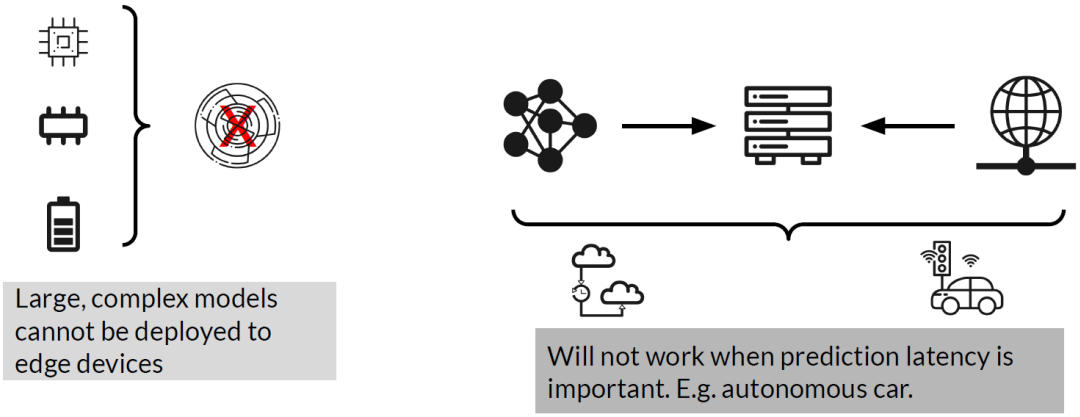
- For example, large companies like Google constantly look for ways to improve resource utilization and reduce costs in applications in data centers using many of the same techniques and technologies that we're discussing in this course.

Constrained Environment: Mobile Phone



- But there are also constraints in distributed environments like mobile phones, Android, and iOS.
- Let's take a look at running a model on a mobile phone, and then take a look at the hardware constraints that these devices can impose.
- In a mobile phone, the average GPU memory size, if you even have a GPU, is much smaller than the one you'll find in a data center, and often **less than about four gigabytes**.
- You'll mostly have **only one GPU**, which is shared by a number of applications and not just yours.
- In most cases, you'll be able to use the GPU for accelerated processing but that comes at a price.
- You have limited GPU available and using it can lead to your **battery draining quickly**.
- Your app will not be received well and could be reviewed poorly if it drains the battery quickly or makes the phone too hot to touch because of complex operations in your ML model.
- There's also **storage limitation** since users don't appreciate large apps using up storage on their phones.

Restrictions in a Constrained Environment



- You can rarely deploy a very large, complex model to a device like a mobile phone.
- If it's too large, users just might choose not to even install it because of the memory constraints.

- Since there are these constraints on memory processing power, battery usage, and all that, there are many classes of models that we simply cannot deploy to mobile phones or embedded systems.
- Instead, we may choose to deploy a model to a server and then expose it through a REST API so that we can use it for inference in our app.
- But of course, this might also not be suitable.
- It might not be feasible to deploy a model to a server in environments where prediction latency is super-important or when a network connection may not always be available.
- One example for this could be an object detection model deployed to an autonomous vehicle.
- It's critical in those applications that the system is able to take actions based on predictions made in near real-time, and it **can't wait for a server round-trip**.

Prediction Latency is Almost Always Important

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

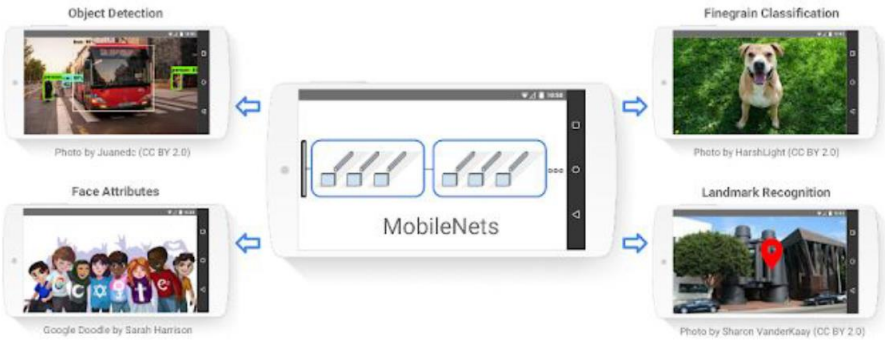
Millisecond turnaround

Model efficiency

Cost

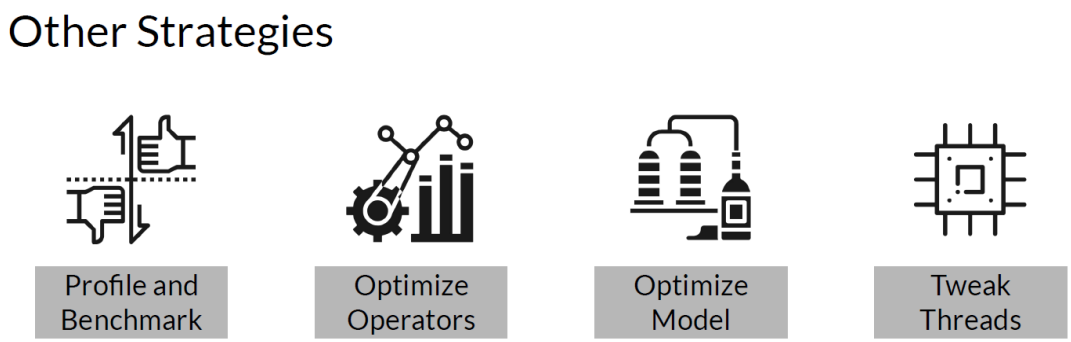
- As a general rule, you should always opt for minimize latency of inference wherever possible.
- This enhances the user experience by reducing the response time of your app, but there are also exceptions.
- Latency might not be as important where it's critical that the model is as accurate as possible, for example, a disease diagnosis.
- You do want to make a trade-off between model complexity, size, accuracy, and prediction latency and understand the cost and constraints of each for the application that you're working on, and that's not an easy task.

Choose Best Model for the Task

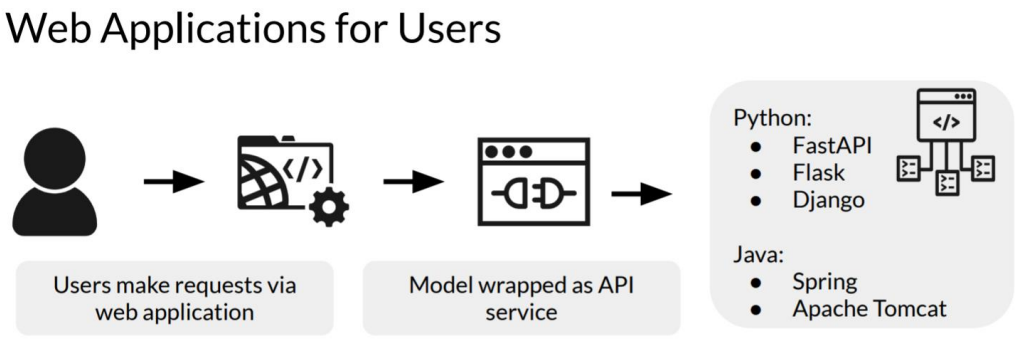


- All of these factors influence your choice of the best model for your task based on your limitations and constraints.
- One example is **MobileNets**, and these are models that are **specifically designed for computer vision on mobile devices.**
- Now they may not have the highest number of predictive classes, and they may not be state of the art in recognition, but all of the work in performing trade-offs for the best mobile model had been done for you already and you can build on this.
- If you're deploying to mobile, you should also follow some strategies for optimizing your model for the constraint mobile environments.

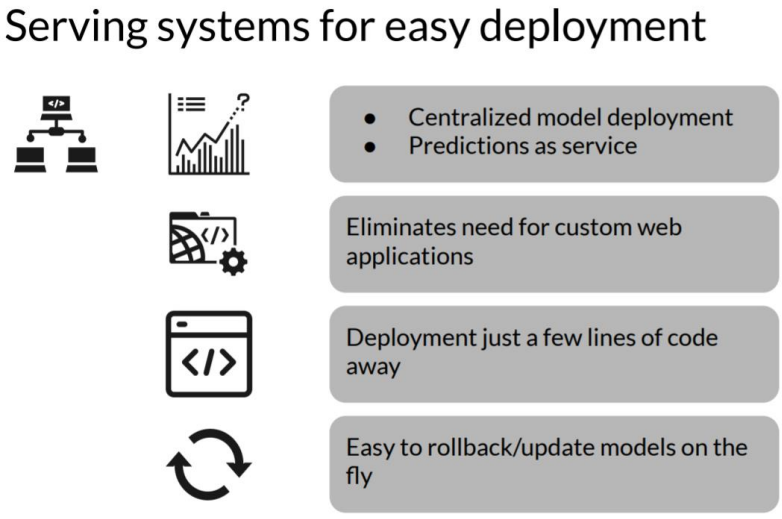
Improving Prediction Latency and Reducing Resource Costs



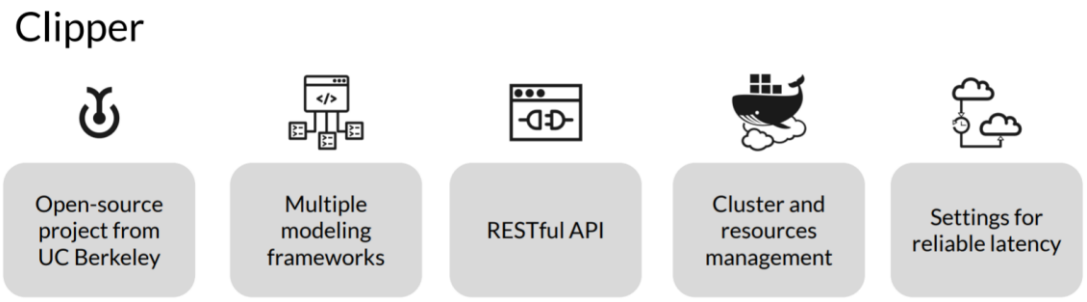
- Once you've selected or built a candidate model, that might be right for your task, it's a good practice to profile and benchmark it.
- The **TensorFlow Lite benchmarking tool** has a built-in profiler that can then show you per operator profiling statistics.
- This can help with understanding performance bottlenecks and identifying which operators dominate the compute time.
- If a particular operator appears frequently in the model and based on profiling, you find that this operator consumes a lot of time and resources, you can look into optimizing it or using a different one.
- We've talked a lot about model optimization, which aims to create smaller models that are generally faster and more energy-efficient.
- This is especially important for deployments on mobile devices.
- TensorFlow Lite supports multiple optimization techniques such as quantization.
- You can also increase the number of interpreter threads to speed up the execution of operators.
- However, increasing the number of threads will also make your model use more resources and power.
- For some applications, latency might be more important than energy efficiency.
- **Multi-threaded execution**, however, also results and increased **performance variability** depending on what else is running concurrently, and this is particularly the case for mobile apps.
- For example, some isolated tests could show a 2x speedup versus a single-threaded version, but if another app is executing at the same time, it can actually result in worse performance than a single-threaded one, so you really need to check it out.



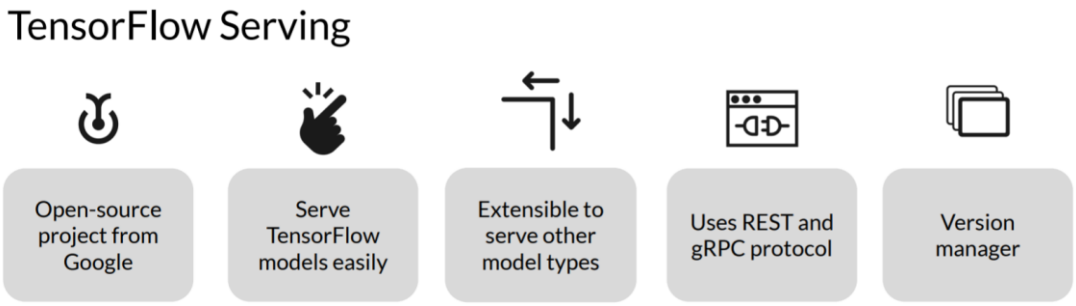
- If you go the other route and deploy a model to a server, there are of course, also some considerations in how you design it.
- The users of your model need a way to make requests, and often this is through a web application.
- The model is wrapped as an API service in this approach, and most **serving infrastructures and languages have web frameworks** that can help you to achieve this.
- For example, Flask is a very popular Python web framework, where it's very easy to roll out an API in Flask.
- If you're familiar with it, you can create a new web client in maybe 10 minutes.
- Django is also a very powerful web framework for Python.
- Similarly, Java also has many options like Apache Tomcat, Spring, et cetera.



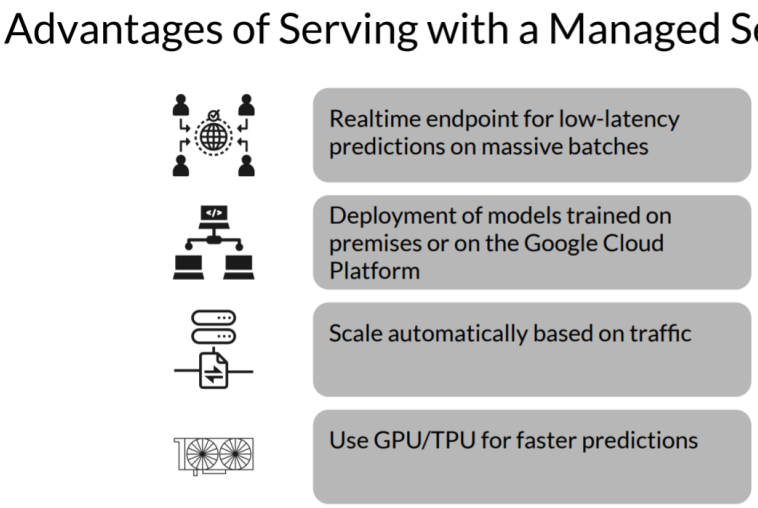
- **Model servers** can manage model deployment. For example, creating the server and managing it to serve prediction requests from clients.
- They eliminate the need for putting models into custom web applications.
- With only a few lines of code, you can generally deploy models.
- They also make it easy to update a rollback models, load, and unload models on demand or when resources are required, and manage multiple versions of models.



- Clipper is a popular **open-source model server** developed at the UC Berkeley Rise Lab.
- Clipper helps you deploy a wide range of models built in frameworks like Caffe, TensorFlow, and Scikit-learn.
- Its overall aim is to be model agnostic.
- Clipper includes a standard REST interface, so this makes it easy for you to integrate with production applications.
- Clipper wraps your models in Docker containers if you want, for cluster and resource management.
- It also helps you set service level objectives for reliable latencies.

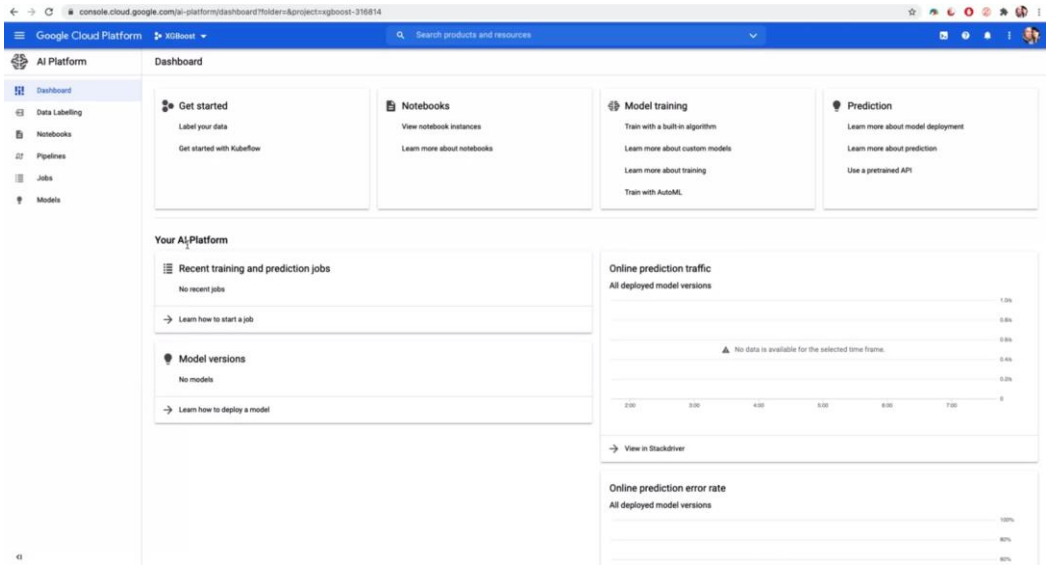


- TensorFlow Serving is also an open-source model server, which offers a flexible high-performance serving system for machine learning models designed for production environments.
- TensorFlow Serving makes it easy to deploy new algorithms in experiments while keeping the same server architecture and APIs.
- TensorFlow Serving provides out of the box integration with TensorFlow models, but it can also be extended to serve other types of models and data.
- TensorFlow Serving offers both the REST and **gRPC** protocols, and **gRPC** is often more efficient than REST.
- TensorFlow Serving has demonstrated performance of up to 100,000 requests per second per core, making it a very powerful tool for serving machine learning applications.
- It has a version manager that can easily load and rollback different versions of the same model and it allows clients to select which version to use for each request.



- There are also advantages to using a managed service to serve your models.
- Let's take a look at one of these called the Google Cloud AI Platform Prediction Service as an example.
- It allows you to set up real-time endpoints which offer low latency predictions, and you can also use it to get predictions on batches of data.
- It also allows you to deploy models that have been trained either on Google Cloud or of course, on your own premises.
- You can scale automatically based on your traffic, which can save you a whole lot of costs, while at the same time giving you that high degree of scalability.
- Of course, there are accelerators available as well, including GPUs and TPUs.
- Other Clouds like Microsoft Azure and Amazon AWS will offer similar capabilities.
- That's a tour of some serving options including mobile web and Cloud.
- Next up, you're going to get hands-on and explore deploying to the AI Prediction Platform. After that, we'll see how to install TensorFlow Serving and then use it to do computer vision predictions.

Creating and Deploying Models to AI Prediction Platform (Lab)



- Let’s go through a Colab for building and deploying an XGBoost model on the Cloud AI platform.
- On the Google Cloud Platform console, scroll down until you see the AI Platform link.

- From here, select Dashboard and you'll be taken to the AI platform dashboard
- Select Models from the list on the left.
- You may see that you need to enable the API to create a model, select that and wait for the progress to complete.
- Once it's done, find the Compute Engine entry in the left navigation menu, and if it isn't enabled to be sure to enable it with this button
- Once it's done, return to the AI platform entry in the navigation and find notebooks.
- Once it opens you'll see a list of notebooks and it's probably empty so go to new instance at the top,
- Select Python3, accept the default and hit Create and you'll see a new entry in the list
- When it's done you'll see a link to open Jupyter lab
- Select it and the Jupyter environment will open up with a number of options, select the Terminal option
- Within the terminal. You can then enter *pip3 install xgboost*
- When it's done you can now open a new python three Notebook. The colab will give you all this code so don't worry if you don't see it right now
- You can run it cell by cell, where the first couple of cells allow you to inspect the data
- As you run through, you'll see how you can create an XGBRegressor for this data which will give you predicted weights versus actual weights for babies.
- You can then save this model as a TensorFlow saved model
- Then the Google Project ID can be found by selecting your project at the top and finding it in the list of projects that you are on.
- The model bucket is a storage bucket that you can name with the model bucket variable, and after you make it, you can see that you can copy the saved model to it
- You can explore your cloud storage to find the bucket, and then within that, you'll see the same model
- Running the code: `gcloud ai-platform models create` will create the AI platform model for you, and entering gcloud ai-platform versions create will deploy a version for you
- When it's done, you can look at your models list and refresh it and you should see that the model is there.
- We have one version called V1, and now we can write some simulated data to adjacent file and use the endpoint to get predictions for it.
- Link to lab above: https://github.com/https-deeplearning-ai/machine-learning-engineering-for-production-public/blob/main/course4/week1-ungraded-labs/C4_W1_Optional_Lab_1_XGBoost_CAIP/C4_W1_Optional_Lab_1.md

Installing 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
```

- Now that you've seen deploying to the Cloud AI platform, you might also want to explore what it would take to deploy to your **own infrastructure**, and the open-source TensorFlow Serving is an ideal solution to that.

- In the next few minutes, we're going to explore how to train a simple model and then deploy and serve it so that you can do remote inferences.
- You'll use the Google Colab environment that runs on Linux. Pretty much everything that you see here you could do on your own server.
- There are several ways to install TensorFlow Serving depending on your needs and your environment. The easiest and most straightforward way of using TensorFlow Serving is with Docker images.
- I **highly recommend this Docker route** unless you have specific needs that are not addressed by running in a container.
- Here's a tip, this is also the easiest way to get TensorFlow Serving working with GPU support.

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

- You can also use TensorFlow Serving using one of the two available **binaries**, the first is simply called TensorFlow-model-server.
- This is a fully optimized server that uses some platform specific compiler optimizations like SSE4 and AVX instructions.
- This should be the preferred options for most users, but it may not work on some older machines.
- For those, you can use a package called TensorFlow-model-server-universal.
- This has been compiled with basic optimizations, but it doesn't include platform specific instruction sets, so it should work on most, if not all machines out there.
- You can use this one if TensorFlow-model-server does not work for you.
- Note that the binary name is the same for both packages, so if you've already installed TensorFlow model server, you should first uninstall it using the apt-get remove TensorFlow model server command.

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

- You can also build TensorFlow Serving from source if you want to fully customize it for your needs.
- Please refer to the documentation for instructions on this.
- On Debian-based Linux, you can install using Aptitude with the apt-get command. That's what you'll be doing in this example.
- In the Colab environment provided, you'll use Aptitude since the VMs that Colabs run on are Debian based.

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
```

- Let's get ready to install TensorFlow Serving using Aptitude since this Colab runs in a Debian environment. You'll add the TensorFlow model server package to the list of packages that Aptitude knows about.
- Note that in Colab you're going to be running as root.
- To install TensorFlow Serving, use *apt-get* install to install the TensorFlow model server package. That's all you need, just that one command line.

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
```

- Now, let's take a look at training a simple computer vision model using the MNIST dataset.
- We'll then serve that for inference using TensorFlow Serving.
- The MNIST dataset contains 70,000 grayscale images of the digit 0-9.
- These images show individual digits at a low resolution, 28 by 28 pixels.
- Even though these are images, we're going to load them as NumPy arrays and not as binary image objects, so we're not dealing with JPEGS or bitmaps or PNGs or anything like that.
- For model training, it's important to **re-scale the pixel intensities** into the range of 0-1, and we call this process **normalization**.

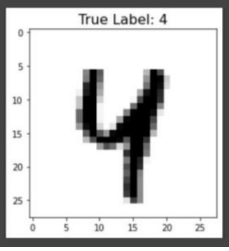
Import the MNIST Dataset

```
# 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
```

- Next, we'll reshape the training set to be an array of 60,000 by 28 by 28 by 1.
- Similarly we'll reshape the test set to be an array of size 10,000 by 28 by 28 by 1.
- The one here is just the size of the color depth of the pixel, and because they're just grayscale, we don't need three channels, red, green, and blue.
- We can print it out now just to check that it's all good.

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()
```



- Before moving forward, it's always important to inspect your data.
- Here we'll visualize one example from the dataset and feel free to change the value of IDX to visualize other examples.

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()
```

- Now let's build a tf.keras.sequential model that can be used to classify the images of the MNIST dataset. We're going to use a very simple CNN.

- Make sure that your model has the correct input shape and the correct number of output units.
- We'll also use softmax for activation in the output layer.

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)
```

- Here, you set your model up for training using the Adam optimizer, sparse categorical cross entropy as the loss, and accuracy for your metrics.
- You'll then train the model for the given number of epochs by fitting training images to the training labels.
- The history object will contain the epoch by epoch metrics for the training.

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
```

- Now we can look at the loss and accuracy on the test set after training, and it's not bad, especially for such a simple model.

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}
```

- For TensorFlow Serving to be able to access the model, you need to save it.

- To do this, you'll use the saved model format.
- This code will save it for you in the directory stored in the MODEL_DIR variable.

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
```

- To let another process such as TensorFlow Serving know where the model is saved, you can use an **environment variable.**
- Here you create one for the model there that you just wrote.
- With TensorFlow Serving installed, you launch it with a bash script.
- Then you use the following parameters to configure the model server.
- The --rest_api_port is the port that requests will be handled on, and in this case it's 8501.
- The –model_name is the name that your model's going to use within the URL and we're using digits_model here.
- The model_base_path should use the environment variable modeled here as the base path to the saved model.

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']
```

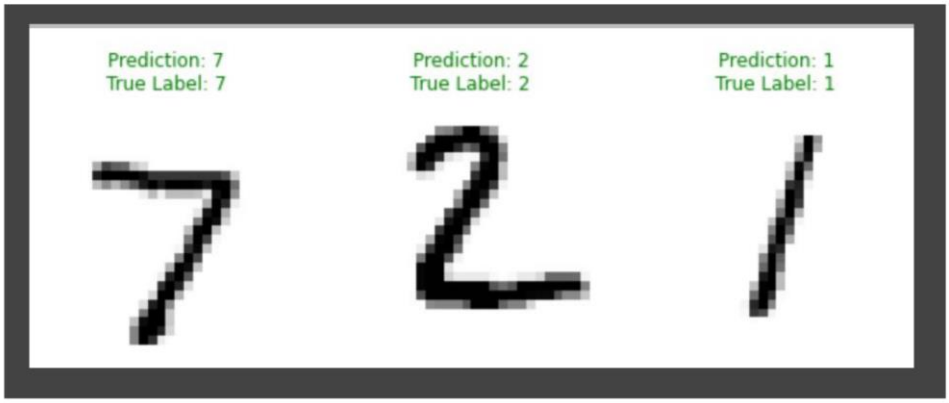
- To send requests for prediction using our saved model, you can use JSON.
- Here you see we create a **JSON object** using the first few images of the test set.
- We'll then send a predict request as a POST to the servers rest endpoints, and you'll pass this a JSON object containing our request data.
- By default, the server will use the latest version of your model, but you could specify a particular version if you needed to here, perhaps for testing or if you want to AB test across multiple users.
- Note the digits_model in the URL, and that's the name that we used in the previous step.

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()
```

- If you want to plot the results, you can do so with code like this.
- Note that if the prediction for a value is the same as for the label, we'll plot it in green, otherwise we'll do it in red.

Results Demo



- Here's the results. Not bad.

TensorFlow Serving – Labs

- TensorFlow Serving with Docker: https://github.com/https-deeplearning-ai/machine-learning-engineering-for-production-public/blob/main/course4/week1-ungraded-labs/C4_W1_Lab_2_TFS_Docker.md
- Serve a model with TensorFlow Serving: https://colab.research.google.com/github/https-deeplearning-ai/machine-learning-engineering-for-production-public/blob/main/course4/week1-ungraded-labs/C4_W1_Lab_3_TFS.ipynb

References

- Docker Intro Lab (GitHub)