Chapter 11. Introduction to Model Serving

This chapter discusses *model serving*—the use of a trained model to generate predictions or results. Also referred to as *running inference*, model serving is the ultimate goal of any trained model.

Training a good ML model is only the first part of the production ML journey. You also need to make your model available to end users or to the business processes that rely on your model's results. Serving it, or including it in an application, is how you make your model available.

NOTE

In the ML space, the words *prediction*, *result*, and *inference* are used somewhat interchangeably.

Model Training

In general, there are two basic types of model training:

Offline training

> The model is trained on a set of already collected data. After deploying to the production environment, the model remains frozen

until it is retrained with new data. The vast majority of model training is offline.

Online training

The model is regularly being updated as new data arrives (e.g., as data streams). This approach is generally limited to cases that use time series data, such as sensor data or stock trading data, to accommodate rapid changes in the data and/or labels. Online training is fairly uncommon and requires unique modeling techniques.

Model Prediction

In general, there are two basic types of model predictions:

Batch predictions

The deployed model makes a set of predictions based on a batch input data containing multiple examples. This is often used when it is not critical to obtain real-time predictions as output.

Real-time predictions (aka on-demand predictions)

Predictions are generated in real time using the input data that is available at the time of the request. This is often used in cases where users or systems are blocked, waiting for the model results.

Unlike model training, where optimizing often refers to improving model prediction metrics, when we discuss optimizing model prediction we are usually concerned with improving model latency, throughput, and cost.

Latency

Latency is the time delay between sending a request to a model and receiving a result. In case of inference, latency includes the whole process of generating a result, from sending data to the model to performing inference and returning the response. Minimal latency, or latency below a certain threshold, is often a key business requirement.

For example, if the latency for online predictions is too long for a travel website, users might complain that an app that suggests hotels is too slow to refresh search results based on the user’s input.

Throughput

Throughput is the number of successful requests served per unit of time, often measured as queries per second (QPS). In some applications, throughput is much more important than latency. Throughput can be thought of as an aggregation of latency.

For example, an offline process might use a model to segment users before storing them in a data warehouse. The goal is to maximize throughput with the least amount of CPU required. Latency for individual requests is not a key concern here, since the application is not customer facing.

Cost

You should always try to minimize the cost associated with each inference, to the extent that the inference still meets the business needs. For a trained model to be viable for a business, the cost to run inference using the model cannot be beyond what the business case justifies.

Accounting for cost includes infrastructure requirements such as the CPU, hardware accelerators such as the GPU, storage and systems to retrieve and supply data, and caches.

There is nearly always a trade-off between cost and performance in terms of latency and/or throughput. Managing this trade-off to meet business and customer needs can be critical to success and is often challenging. It also often changes over the life of an application and needs to be revisited regularly.

In applications where latency and throughput can suffer slightly, you can reduce costs by using strategies such as sharing GPUs among multiple models and performing multimodel serving.

Resources and Requirements for Serving Models

There are good reasons why models often become complex, and some not-so-good reasons too. Sometimes it's because the nature of the problem means they need to model more complex relationships. That's a perfectly valid and necessary reason to add complexity. A not-so-valid reason is that there is a natural impulse to apply the latest hot, new, complex model architectures because, well, they're pretty cool. Another not-so-valid reason is a sort of lazy impulse to include more and more features on the assumption that more is better.

Whether the reason is valid or not, adding model complexity often results in longer prediction latencies and/or higher infrastructure costs. But if it is applied correctly, added complexity can also lead to a boost in prediction accuracy.

Cost and Complexity

As models become more complex and/or more and more features are included, the resource requirements increase for every part of the training and serving infrastructure. Increased resource requirements result in increased cost and increased hardware requirements, along with management of larger model registries, which results in a higher support and maintenance burden.

As with many things in life, the key is to find the right balance. Finding the right balance between cost and complexity is a skill that seasoned practitioners build over time.

There’s also usually a trade-off between the model’s predictive effectiveness and the speed of its prediction latency. Depending on the use case, you need to decide on two metrics:

- The model’s optimizing metric, which reflects the model’s predictive effectiveness. Examples include accuracy, precision, and mean square error. The better the value of this metric, the better the model.
- The model’s gating metric, which reflects an operational constraint the model needs to satisfy, such as prediction latency. For example, you might set a latency threshold to a particular value, such as 200 milliseconds, and any model that doesn’t meet the threshold is not accepted. Another example of a gating metric is the size of the model, which is important if you plan to deploy your model to low-spec hardware such as mobile and embedded devices.

One approach to making the necessary choices and balancing these trade-offs is to specify the serving infrastructure (CPU, GPU, TPU), and start increasing your model complexity (if and only if it improves your model predictive power) until you hit one or more of your gating metrics on that infrastructure. Then, assess the results, and either accept the model as it is, work to improve its accuracy and/or reduce its complexity, or make the decision to increase the specifications of the serving infrastructure.

Accelerators

One of the factors to consider when designing your serving and training infrastructure is the use of accelerators, such as GPUs and TPUs. Each has different advantages, costs, and limitations.

GPUs tend to be optimized for parallel throughput and are often used in training infrastructure, while TPUs have advantages for large, complex models and large batch sizes, especially for inference. These decisions have significant effects on a project's budget. There is also a trade-off between applying a larger number of less powerful accelerators and using a smaller number of more powerful accelerators. A larger number of less powerful accelerators can be more resilient to failures, be more scalable at smaller granularities, and may or may not be more cost efficient, but it also increases the complexity of distribution and requires smaller shards.

Often when working with a team or department, these choices need to be made for a broad range of models, and not just the new model you're working on now, because these are shared resources.

Feeding the Beast

The prediction request to your ML model might not provide all the features required for prediction. Some of the features might be precomputed or aggregated, and read in real time from a datastore.

Take the example of a food delivery app that should predict the estimated time for order delivery. This is based on a number of features such as the list of incoming orders and the number of outstanding orders per minute in the past hour. Features such as these will be read from a datastore. You will need powerful caches to retrieve this data with low latency, since delivery time has to be updated in real time. You cannot wait for seconds to retrieve data from a database. So, of course, this has cost implications.

NoSQL databases are a good solution to implement caching and feature lookup. Various options are available:

- If you need submillisecond read latency on a limited amount of quickly changing data retrieved by a few thousand clients, one good choice is Google Cloud Memorystore. It’s a fully managed version of Redis and Memcache, which are also good open source options.
- If you need millisecond read latency on slowly changing data where storage scales automatically, one good choice is 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 is Google Cloud Bigtable.
- Amazon DynamoDB is a good choice for a scalable, low read latency database with an in-memory cache.

Adding caches speeds up feature lookup and prediction retrieval latency. You have to carefully choose from the different available offerings based on your requirements, and balance that with your budget constraints.

Model Deployments

When deciding where to deploy a model, you primarily have two choices:

- You can have a centralized model in a data center that is accessed by a remote call.
- Or you can distribute instances of your model to devices that are closer to the end user, such as in a mobile, edge, or embedded system deployment.

Data Center Deployments

Cost and efficiency are important at any scale, even when you have large resources in a huge data center. For example, Google constantly looks for ways to improve its resource utilization and reduce costs in its applications and data centers, using many of the same techniques and technologies discussed in the chapters that follow.

Data center deployments are typically far less resource constrained than mobile deployments, because you have whole servers or clusters of servers at your disposal in a high-bandwidth networked environment. That doesn’t mean you want to waste expensive resources, and you also need to account for uneven demand for your model by including server scaling as a key factor when designing your serving infrastructure. When serving online, your infrastructure needs to be able to scale up to a level just higher than your peak demand and scale down to a level just higher than your minimum demand—usually while keeping your model ready to respond to requests with acceptable latency.

We will explore many of the serving scenarios and techniques that apply to serving in data centers in the chapters that follow.

Mobile and Distributed Deployments

Let’s look at running a model as part of an app on a mobile phone and discuss the hardware constraints these devices impose.

In a budget mobile phone, the average GPU memory size is less than 4 GB. You will mostly have only one GPU, which is shared by a number of applications, not just your model. Even now, some phones don’t even have GPUs. In most cases, you will be able to use the GPU for accelerated processing, but that comes with a price. You have limited GPUs available, and using the GPU might lead to your battery draining quickly. Your app will not be received well if it drains the battery quickly, or if it makes the phone too hot to touch because of complex operations in your ML model.

There are also storage limitations, since users don’t appreciate large apps using up the storage on their phones. You can rarely deploy a very large, complex model on a device such as a mobile phone or camera. If it’s too large, users might choose not to install your app because of memory constraints.

So instead, you may choose to deploy your model on a server (usually in a data center as discussed in the preceding section, but really wherever you can run your server) and serve requests through a REST API so that you can use it for inference in an app.

This may not be an issue in models used in face filter apps, object detection, age detection, and other entertainment purposes. But it isn’t feasible to deploy on a server in environments where prediction latency is important or when a network connection may not always be available. One example is models for object detection deployed on autonomous vehicles. It’s critical in those applications that the system is able to take actions based on predictions made in real time, so relying on a connection to a central data center is not a viable option.

As a general rule, you should always opt for on-device inference whenever possible. This enhances the user experience by reducing the response time of your app.

But there are also exceptions. Latency may not be as important when it’s critical that the model is as accurate as possible. So you need to make a trade-off between model complexity, size, accuracy, and prediction latency and understand the costs and constraints of each for the application you’re working on. All of these factors influence your choice of the best model for your task, based on your limitations and constraints. For example, you may want to choose one of the MobileNet models, which are models optimized for mobile vision applications.

Once you have selected a candidate model that may be right for your task, it’s a good practice to profile and benchmark it. The TensorFlow Lite (TF Lite) benchmarking tool has a built-in profiler that shows per-operator profiling statistics. This can help you understand performance bottlenecks and identify which operators dominate the compute time. If a particular operator appears frequently in the model and, based on profiling, you find that the operator consumes a lot of time, you can look into optimizing that operator.

We previously discussed 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. TF 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 make your model use more resources and power. For some applications, latency may be more important than energy efficiency. Multithreaded execution, however, also results in increased performance variability depending on what else is running concurrently. This is particularly the case for mobile apps. For example, isolated tests may show a 2x speedup over single-threaded execution, but if another app is executing at the same time, it may actually result in lower performance than single-threaded execution.

Model Servers

Users of your model need a way to make requests. Often this is through a web application that makes calls to a server hosting your model. The model is wrapped as an API service in this approach.

Both Python and Java have many web frameworks that can help you achieve this. For example, Flask is a very popular Python web framework. It’s very easy to create an API in Flask; if you are familiar with Flask, you can create a new web client in about 10 minutes. Django is also a very powerful web framework in Python. Similarly, Java has many options, including Apache Tomcat and Spring.

Model servers such as TensorFlow Serving can manage model deployment; for example, creating the server instance and managing it to serve prediction requests from clients. These model servers eliminate the need to put models into custom web applications. They also make it easy to update/roll back models, load and unload models on demand or when resources are required, and manage multiple versions of models.

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

Clipper is a popular open source model server developed at the UC Berkeley RISE Lab. Clipper helps you deploy a wide range of model frameworks, including Caffe, TensorFlow, and scikit-learn. It aims to be model agnostic, and it includes a standard REST interface, which makes it easy to integrate with production applications. Clipper wraps your models in Docker containers for cluster and resource management. It also allows you to set service-level objectives for reliable latencies.

Managed Services

Managed services are another option for serving your models. There are several advantages to using a managed service to serve your models.

Google Cloud Vertex AI is a managed service that allows you to set up real-time endpoints that offer low-latency predictions. You can also use it to get predictions on batches of data. In addition, Vertex AI allows you to deploy models that have been trained either in the cloud or anywhere else. And you can scale automatically based on your traffic, which can save you a lot of cost but at the same time give you a high degree of scalability. There are accelerators available as well, including GPUs and TPUs. Microsoft Azure and Amazon AWS also offer managed services with similar capabilities.

Conclusion

This chapter provided an introduction to model serving, which we'll continue discussing in the next three chapters. Model serving is a very important part of production ML, and in many cases it is the largest contributor to the cost of using ML in a product or service, so having a good understanding of the issues and techniques of model serving is important.