Chapter 7. Model Deployment and Prediction Service

In Chapters 4 through 6, we have discussed the considerations for developing an ML model, from creating training data, extracting features, and developing the model to crafting metrics to evaluate this model. These considerations constitute the logic of the model—instructions on how to go from raw data into an ML model, as shown in Figure 7-1. Developing this logic requires both ML knowledge and subject matter expertise. In many companies, this is the part of the process that is done by the ML or data science teams.

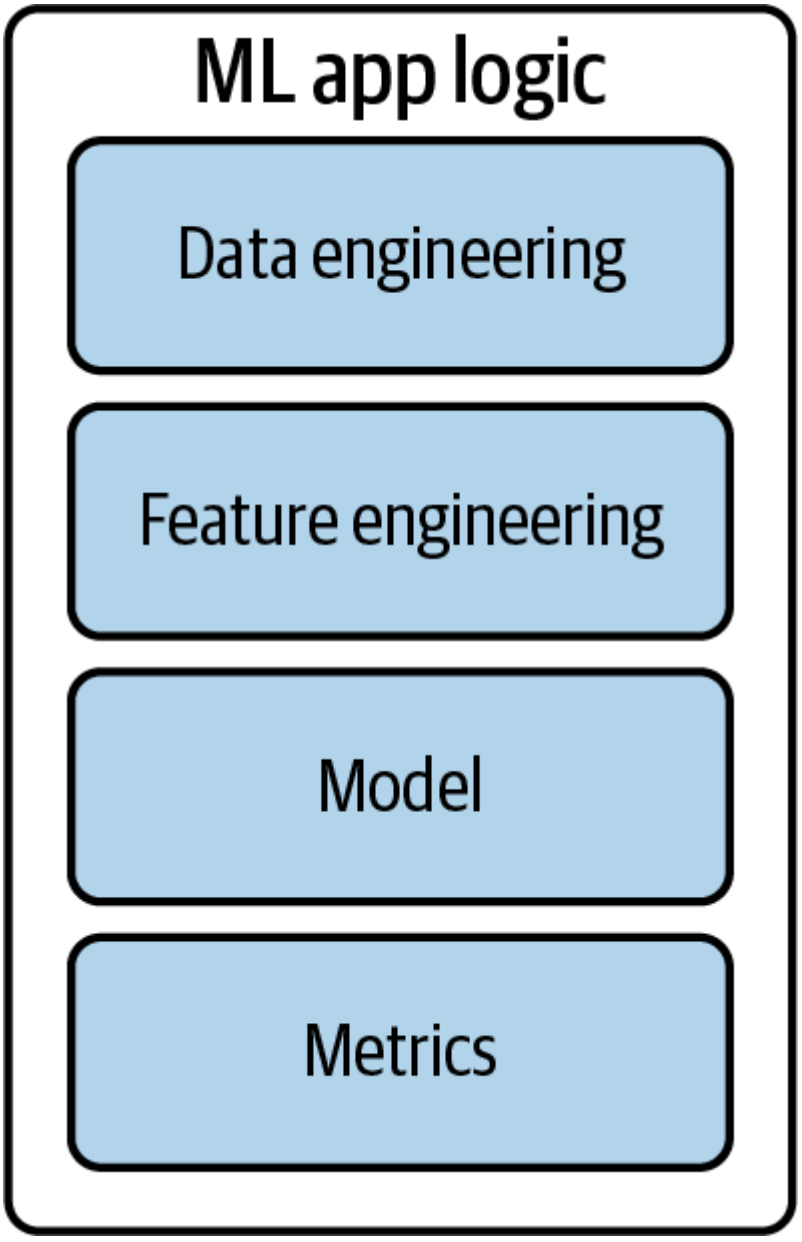

Figure 7-1. Different aspects that make up the ML model logic

In this chapter, we'll discuss another part in the iterative process: deploying your model. "Deploy" is a loose term that generally means making your model running and accessible. During model development, your model usually runs in a development environment.[1] To be deployed, your model will have to leave the development environment. Your model can be deployed to a staging environment for testing or to a production environment to be used by your end users. In this chapter, we focus on deploying models to production environments.

Before we move forward, I want to emphasize that production is a spectrum. For some teams, production means generating nice plots in notebooks to show to the business team. For other teams, production means keeping your models up and running for millions of users a day. If your

work is in the first scenario, your production environment is similar to the development environment, and this chapter is less relevant for you. If your work is closer to the second scenario, read on.

I once read somewhere on the internet: deploying is easy if you ignore all the hard parts. If you want to deploy a model for your friends to play with, all you have to do is to wrap your predict function in a POST request endpoint using Flask or FastAPI, put the dependencies this predict function needs to run in a container,[2] and push your model and its associated container to a cloud service like AWS or GCP to expose the endpoint:

```
# Example of how to use FastAPI to turn your predict function
# into a POST endpoint
@app.route('/predict', methods=['POST'])
def predict():
    X = request.get_json()['X']
    y = MODEL.predict(X).tolist()
    return json.dumps({'y': y}), 200
```

You can use this exposed endpoint for downstream applications: e.g., when an application receives a prediction request from a user, this request is sent to the exposed endpoint, which returns a prediction. If you're familiar with the necessary tools, you can have a functional deployment in an hour. My students, after a 10-week course, were all able to deploy an ML application as their final projects even though few have had deployment experience before.[3]

The hard parts include making your model available to millions of users with a latency of milliseconds and 99% uptime, setting up the infrastructure so that the right person can be immediately notified when something goes wrong, figuring out what went wrong, and seamlessly deploying the updates to fix what's wrong.

In many companies, the responsibility of deploying models falls into the hands of the same people who developed those models. In many other companies, once a model is ready to be deployed, it will be exported and handed off to another team to deploy it. However, this separation of responsibilities can cause high overhead communications across teams and

make it slow to update your model. It also can make it hard to debug should something go wrong. We'll discuss more on team structures in Chapter 11.

NOTE

Exporting a model means converting this model into a format that can be used by another application. Some people call this process "serialization."[4] There are two parts of a model that you can export: the model definition and the model's parameter values. The model definition defines the structure of your model, such as how many hidden layers it has and how many units in each layer. The parameter values provide the values for these units and layers. Usually, these two parts are exported together.

In TensorFlow 2, you might use `tf.keras.Model.save()` to export your model into TensorFlow's SavedModel format. In PyTorch, you might use `torch.onnx.export()` to export your model into ONNX format.

Regardless of whether your job involves deploying ML models, being cognizant of how your models are used can give you an understanding of their constraints and help you tailor them to their purposes.

In this chapter, we'll start off with some common myths about ML deployment that I've often heard from people who haven't deployed ML models. We'll then discuss the two main ways a model generates and serves its predictions to users: online prediction and batch prediction. The process of generating predictions is called *inference*.

We'll continue with where the computation for generating predictions should be done: on the device (also referred to as the edge) and the cloud. How a model serves and computes the predictions influences how it should be designed, the infrastructure it requires, and the behaviors that users encounter.

If you come from an academic background, some of the topics discussed in this chapter might be outside your comfort zone. If an unfamiliar term comes up, take a moment to look it up. If a section becomes too dense,

feel free to skip it. This chapter is modular, so skipping a section shouldn't affect your understanding of another section.

Machine Learning Deployment Myths

As discussed in Chapter 1, deploying an ML model can be very different from deploying a traditional software program. This difference might cause people who have never deployed a model before to either dread the process or underestimate how much time and effort it will take. In this section, we'll debunk some of the common myths about the deployment process, which will, hopefully, put you in a good state of mind to begin the process. This section will be most helpful to people with little to no deploying experience.

Myth 1: You Only Deploy One or Two ML Models at a Time

When doing academic projects, I was advised to choose a small problem to focus on, which usually led to a single model. Many people from academic backgrounds I've talked to tend to also think of ML production in the context of a single model. Subsequently, the infrastructure they have in mind doesn't work for actual applications, because it can only support one or two models.

In reality, companies have many, many ML models. An application might have many different features, and each feature might require its own model. Consider a ride-sharing app like Uber. It needs a model to predict each of the following elements: ride demand, driver availability, estimated time of arrival, dynamic pricing, fraudulent transaction, customer churn, and more. Additionally, if this app operates in 20 countries, until you can have models that generalize across different user-profiles, cultures, and languages, each country would need its own set of models. So with 20 countries and 10 models for each country, you already have 200 models. Figure 7-2 shows a wide range of the tasks that leverage ML at Netflix.

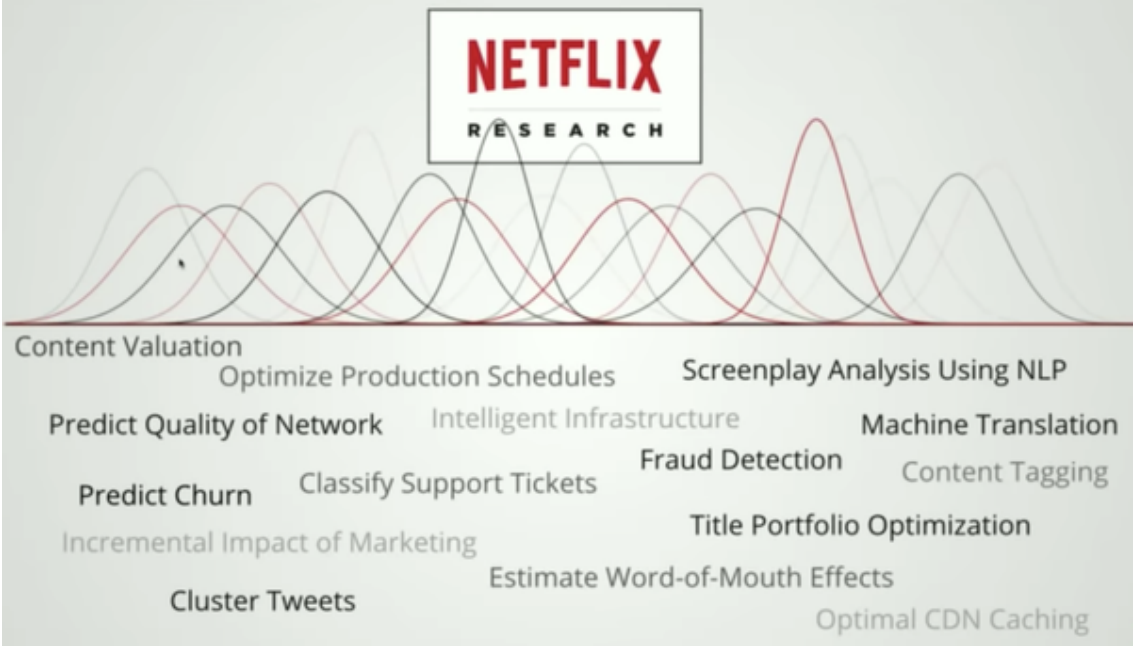

Figure 7-2. Different tasks that leverage ML at Netflix. Source: Ville Tuulos[5]

In fact, Uber has thousands of models in production.[6] At any given moment, Google has thousands of models training concurrently with hundreds of billions parameters in size.[7] Booking.com has 150+ models.[8] A 2021 study by Algorithmia shows that among organizations with over 25,000 employees, 41% have more than 100 models in production.[9]

Myth 2: If We Don't Do Anything, Model Performance Remains the Same

Software doesn't age like fine wine. It ages poorly. The phenomenon in which a software program degrades over time even if nothing seems to have changed is known as "software rot" or "bit rot."

ML systems aren't immune to it. On top of that, ML systems suffer from what are known as data distribution shifts, when the data distribution your model encounters in production is different from the data distribution it was trained on.[10] Therefore, an ML model tends to perform best right after training and to degrade over time.

Myth 3: You Won't Need to Update Your Models as Much

People tend to ask me: "How often *should* I update my models?" It's the wrong question to ask. The right question should be: "How often *can* I update my models?"

Since a model's performance decays over time, we want to update it as fast as possible. This is an area of ML where we should learn from existing DevOps best practices. Even back in 2015, people were already constantly pushing out updates to their systems. Etsy deployed 50 times/day, Netflix thousands of times per day, AWS every 11.7 seconds.[11]

While many companies still only update their models once a month, or even once a quarter, Weibo's iteration cycle for updating some of their ML models is 10 minutes.[12] I've heard similar numbers at companies like Alibaba and ByteDance (the company behind TikTok).

In the words of Josh Wills, a former staff engineer at Google and director of data engineering at Slack, "We're always trying to bring new models into production just as fast as humanly possible."[13]

We'll discuss more on the frequency to retrain your models in Chapter 9.

Myth 4: Most ML Engineers Don't Need to Worry About Scale

What "scale" means varies from application to application, but examples include a system that serves hundreds of queries per second or millions of users a month.

You might argue that, if so, only a small number of companies need to worry about it. There is only one Google, one Facebook, one Amazon. That's true, but a small number of large companies employ the majority of the software engineering workforce. According to the Stack Overflow Developer Survey 2019, more than half of the respondents worked for a company of at least 100 employees (see Figure 7-3). This isn't a perfect correlation, but a company of 100 employees has a good chance of serving a reasonable number of users.

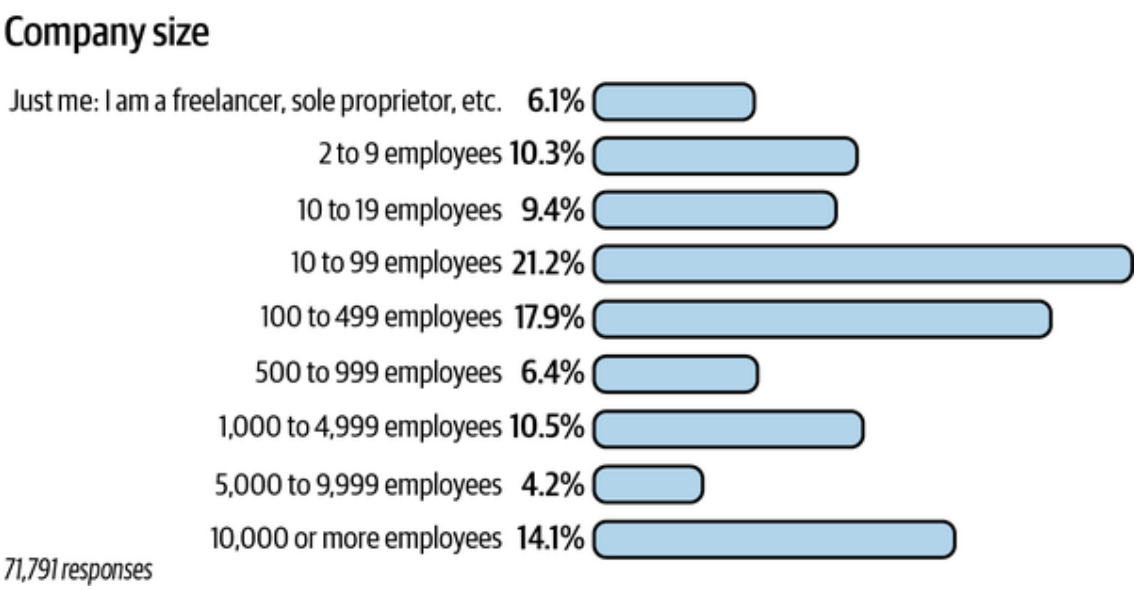

Figure 7-3. The distribution of the size of companies where software engineers work. Source: Adapted from an image by Stack Overflow[14]

I couldn't find a survey for ML-specific roles, so I asked on Twitter and found similar results. This means that if you're looking for an ML-related job in the industry, you'll likely work for a company of at least 100 employees, whose ML applications likely need to be scalable. Statistically speaking, an ML engineer should care about scale.

Batch Prediction Versus Online Prediction

One fundamental decision you'll have to make that will affect both your end users and developers working on your system is how it generates and serves its predictions to end users: online or batch. The terminologies surrounding batch and online prediction are still quite confusing due to the lack of standardized practices in the industry. I'll do my best to explain the nuances of each term in this section. If you find any of the terms mentioned here too confusing, feel free to ignore them for now. If you forget everything else, there are three main modes of prediction that I hope you'll remember:

- Batch prediction, which uses only batch features.
- Online prediction that uses only batch features (e.g., precomputed embeddings).

- Online prediction that uses both batch features and streaming features. This is also known as streaming prediction.

Online prediction is when predictions are generated and returned as soon as requests for these predictions arrive. For example, you enter an English sentence into Google Translate and get back its French translation immediately. Online prediction is also known as *on-demand prediction*. Traditionally, when doing online prediction, requests are sent to the prediction service via RESTful APIs (e.g., HTTP requests—see "Data Passing Through Services"). When prediction requests are sent via HTTP requests, online prediction is also known as *synchronous prediction*: predictions are generated in synchronization with requests.

Batch prediction is when predictions are generated periodically or whenever triggered. The predictions are stored somewhere, such as in SQL tables or an in-memory database, and retrieved as needed. For example, Netflix might generate movie recommendations for all of its users every four hours, and the precomputed recommendations are fetched and shown to users when they log on to Netflix. Batch prediction is also known as *asynchronous prediction*: predictions are generated asynchronously with requests.

TERMINOLOGY CONFUSION

The terms "online prediction" and "batch prediction" can be confusing. Both can make predictions for multiple samples (in batch) or one sample at a time. To avoid this confusion, people sometimes prefer the terms "synchronous prediction" and "asynchronous prediction." However, this distinction isn't perfect either, because when online prediction leverages a real-time transport to send prediction requests to your model, the requests and predictions technically are asynchronous.

Figure 7-4 shows a simplified architecture for batch prediction, and Figure 7-5 shows a simplified version of online prediction using only batch features. We'll go over what it means to use only batch features next.

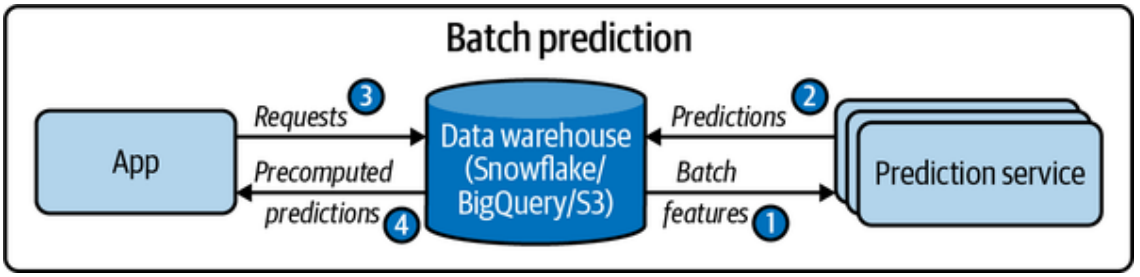

Figure 7-4. A simplified architecture for batch prediction

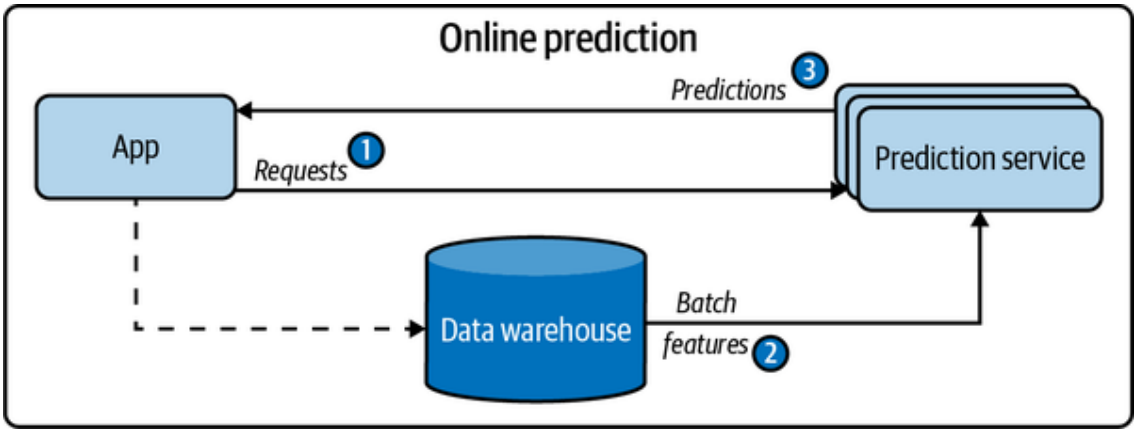

Figure 7-5. A simplified architecture for online prediction that uses only batch features

As discussed in Chapter 3, features computed from historical data, such as data in databases and data warehouses, are *batch features*. Features computed from streaming data—data in real-time transports—are *streaming features*. In batch prediction, only batch features are used. In online prediction, however, it's possible to use both batch features and streaming features. For example, after a user puts in an order on DoorDash, they might need the following features to estimate the delivery time:

Batch features

: The mean preparation time of this restaurant in the past

Streaming features

: In the last 10 minutes, how many other orders they have, and how many delivery people are available

STREAMING FEATURES VERSUS ONLINE FEATURES

I've heard the terms "streaming features" and "online features" used interchangeably. They are actually different. Online features are more general, as they refer to any feature used for online prediction, including batch features stored in memory.

A very common type of batch feature used for online prediction, especially session-based recommendations, is item embeddings. Item embeddings are usually precomputed in batch and fetched whenever they are needed for online prediction. In this case, embeddings can be considered online features but not streaming features.

Streaming features refer exclusively to features computed from streaming data.

A simplified architecture for online prediction that uses both streaming features and batch features is shown in Figure 7-6. Some companies call this kind of prediction "streaming prediction" to distinguish it from the kind of online prediction that doesn't use streaming features.

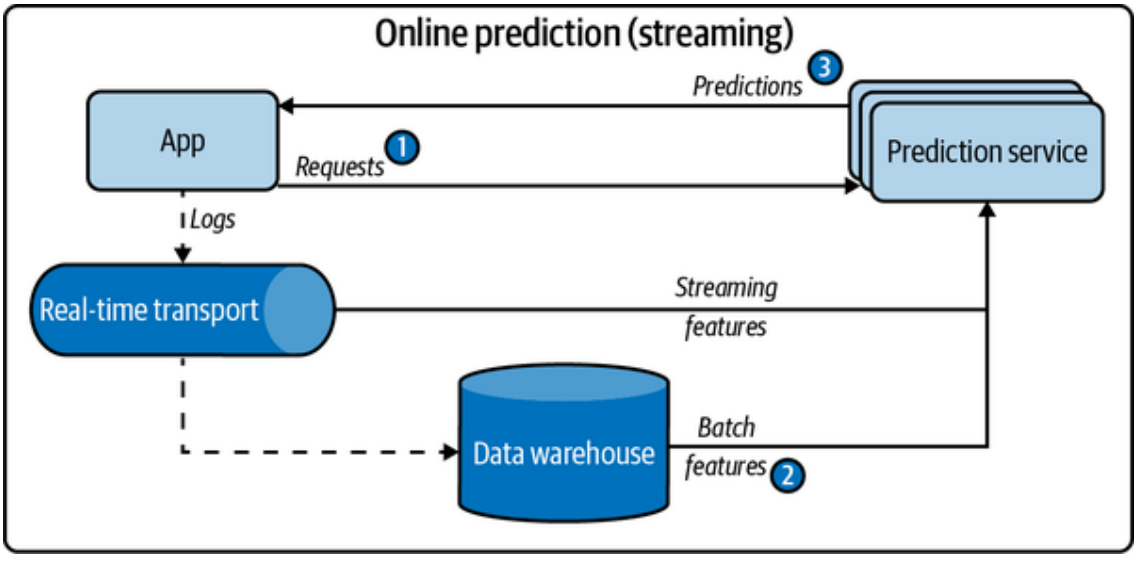

Figure 7-6. A simplified architecture for online prediction that uses both batch features and streaming features

However, online prediction and batch prediction don't have to be mutually exclusive. One hybrid solution is that you precompute predictions for popular queries, then generate predictions online for less popular queries. Table 7-1 summarizes the key points to consider for online prediction and batch prediction.

Table 7-1. Some key differences between batch prediction and online prediction

	Batch prediction (asynchronous)	Online prediction (synchronous)
Frequency	Periodical, such as every four hours	As soon as requests come
Useful for	Processing accumulated data when you don't need immediate results (such as recommender systems)	When predictions are needed as soon as a data sample is generated (such as fraud detection)
Optimized for	High throughput	Low latency

In many applications, online prediction and batch prediction are used side by side for different use cases. For example, food ordering apps like DoorDash and UberEats use batch prediction to generate restaurant recommendations—it'd take too long to generate these recommendations online because there are many restaurants. However, once you click on a restaurant, food item recommendations are generated using online prediction.

Many people believe that online prediction is less efficient, both in terms of cost and performance, than batch prediction because you might not be able to batch inputs together and leverage vectorization or other optimization techniques. This is not necessarily true, as we already discussed in the section "Batch Processing Versus Stream Processing".

Also, with online prediction, you don't have to generate predictions for users who aren't visiting your site. Imagine you run an app where only 2% of your users log in daily—e.g., in 2020, Grubhub had 31 million users and 622,000 daily orders.[15] If you generate predictions for every user each day, the compute used to generate 98% of your predictions will be wasted.

From Batch Prediction to Online Prediction

To people coming to ML from an academic background, the more natural way to serve predictions is probably online. You give your model an input and it generates a prediction as soon as it receives that input. This is likely how most people interact with their models while prototyping. This is also likely easier to do for most companies when first deploying a model. You export your model, upload the exported model to Amazon SageMaker or Google App Engine, and get back an exposed endpoint.[16] Now, if you send a request that contains an input to that endpoint, it will send back a prediction generated on that input.

A problem with online prediction is that your model might take too long to generate predictions. Instead of generating predictions as soon as they arrive, what if you compute predictions in advance and store them in your database, and fetch them when requests arrive? This is exactly what batch prediction does. With this approach, you can generate predictions for multiple inputs at once, leveraging distributed techniques to process a high volume of samples efficiently.

Because the predictions are precomputed, you don't have to worry about how long it'll take your models to generate predictions. For this reason, batch prediction can also be seen as a trick to reduce the inference latency of more complex models—the time it takes to retrieve a prediction is usually less than the time it takes to generate it.

Batch prediction is good for when you want to generate a lot of predictions and don't need the results immediately. You don't have to use all the predictions generated. For example, you can make predictions for all customers on how likely they are to buy a new product, and reach out to the top 10%.

However, the problem with batch prediction is that it makes your model less responsive to users' change preferences. This limitation can be seen even in more technologically progressive companies like Netflix. Say you've been watching a lot of horror movies lately, so when you first log in to Netflix, horror movies dominate recommendations. But you're feel-

ing bright today, so you search “comedy” and start browsing the comedy category. Netflix should learn and show you more comedy in your list of their recommendations, right? As of writing this book, it can’t update the list until the next batch of recommendations is generated, but I have no doubt that this limitation will be addressed in the near future.

Another problem with batch prediction is that you need to know what requests to generate predictions for in advance. In the case of recommending movies for users, you know in advance how many users to generate recommendations for.[17] However, for cases when you have unpredictable queries—if you have a system to translate from English to French, it might be impossible to anticipate every possible English text to be translated—you need to use online prediction to generate predictions as requests arrive.

In the Netflix example, batch prediction causes mild inconvenience (which is tightly coupled with user engagement and retention), not catastrophic failures. There are many applications where batch prediction would lead to catastrophic failures or just wouldn’t work. Examples where online prediction is crucial include high-frequency trading, autonomous vehicles, voice assistants, unlocking your phone using face or fingerprints, fall detection for elderly care, and fraud detection. Being able to detect a fraudulent transaction that happened three hours ago is still better than not detecting it at all, but being able to detect it in real time can prevent the fraudulent transaction from going through.

Batch prediction is a workaround for when online prediction isn’t cheap enough or isn’t fast enough. Why generate one million predictions in advance and worry about storing and retrieving them if you can generate each prediction as needed at the exact same cost and same speed?

As hardware becomes more customized and powerful and better techniques are being developed to allow faster, cheaper online predictions, online prediction might become the default.

In recent years, companies have made significant investments to move from batch prediction to online prediction. To overcome the latency challenge of online prediction, two components are required:

- A (near) real-time pipeline that can work with incoming data, extract streaming features (if needed), input them into a model, and return a prediction in near real time. A streaming pipeline with real-time transport and a stream computation engine can help with that.
- A model that can generate predictions at a speed acceptable to its end users. For most consumer apps, this means milliseconds.

We've discussed stream processing in Chapter 3. We'll continue discussing the unification of the stream pipeline with the batch pipeline in the next section. Then we'll discuss how to speed up inference in the section "Model optimization".

Unifying Batch Pipeline and Streaming Pipeline

Batch prediction is largely a product of legacy systems. In the last decade, big data processing has been dominated by batch systems like MapReduce and Spark, which allow us to periodically process a large amount of data very efficiently. When companies started with ML, they leveraged their existing batch systems to make predictions. When these companies want to use streaming features for their online prediction, they need to build a separate streaming pipeline. Let's go through an example to make this more concrete.

Imagine you want to build a model to predict arrival time for an application like Google Maps. The prediction is continually updated as a user's trip progresses. A feature you might want to use is the average speed of all the cars in your path in the last five minutes. For training, you might use data from the last month. To extract this feature from your training data, you might want to put all your data into a dataframe to compute this feature for multiple training samples at the same time. During inference, this feature will be continually computed on a sliding window. This means that in training this feature is computed in batch, whereas during inference this feature is computed in a streaming process.

Having two different pipelines to process your data is a common cause for bugs in ML production. One cause for bugs is when the changes in one pipeline aren't correctly replicated in the other, leading to two pipelines

extracting two different sets of features. This is especially common if the two pipelines are maintained by two different teams, such as the ML team maintains the batch pipeline for training while the deployment team maintains the stream pipeline for inference, as shown in Figure 7-7.

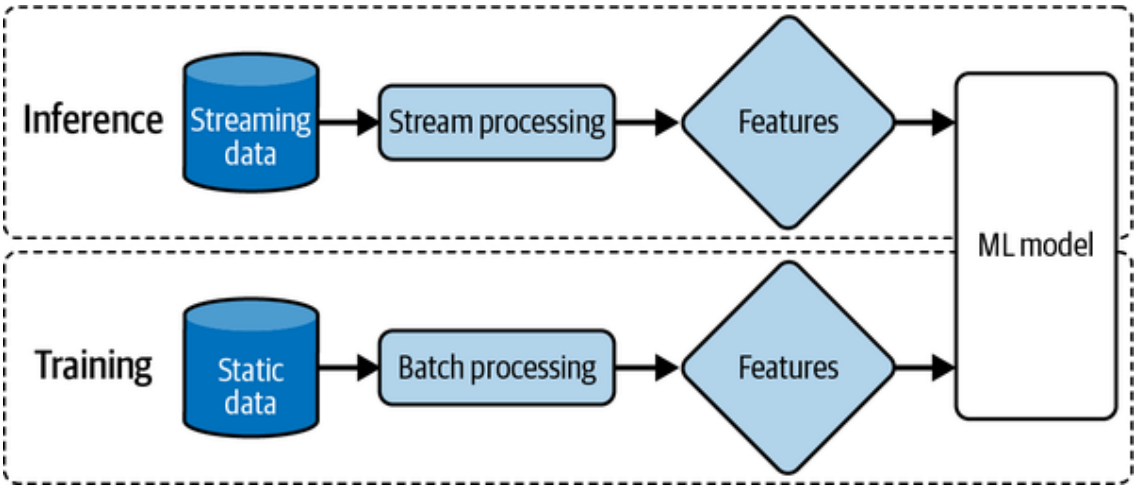

Figure 7-7. Having two different pipelines for training and inference is a common source for bugs for ML in production

Figure 7-8 shows a more detailed but also more complex feature of the data pipeline for ML systems that do online prediction. The boxed element labeled Research is what people are often exposed to in an academic environment.

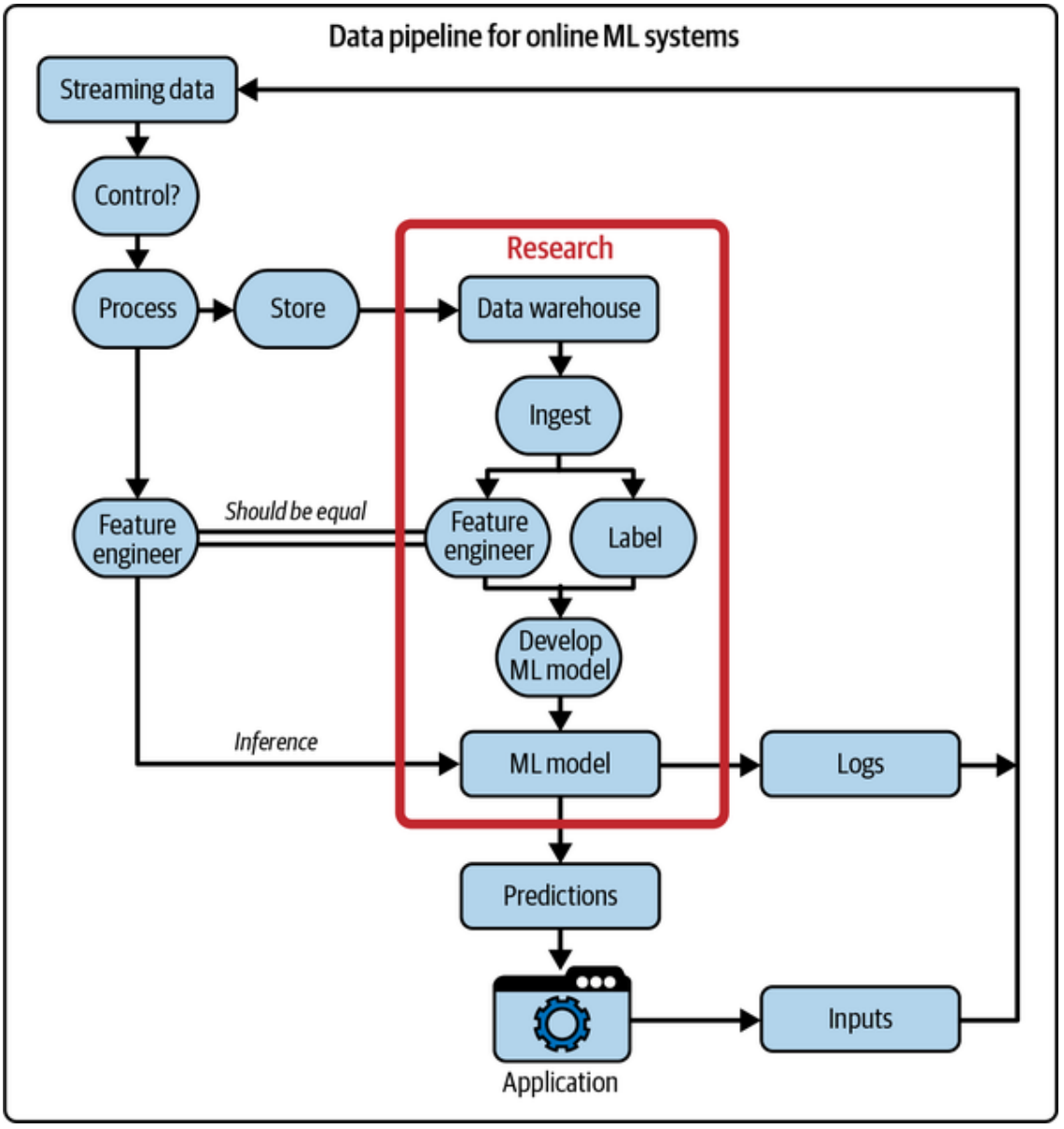

Figure 7-8. A data pipeline for ML systems that do online prediction

Building infrastructure to unify stream processing and batch processing has become a popular topic in recent years for the ML community. Companies including Uber and Weibo have made major infrastructure overhauls to unify their batch and stream processing pipelines by using a stream processor like Apache Flink.[18] Some companies use feature stores to ensure the consistency between the batch features used during training and the streaming features used in prediction. We'll discuss feature stores in Chapter 10.

Model Compression

We've talked about a streaming pipeline that allows an ML system to extract streaming features from incoming data and input them into an ML model in (near) real time. However, having a near (real-time) pipeline isn't enough for online prediction. In the next section, we'll discuss techniques for fast inference for ML models.

If the model you want to deploy takes too long to generate predictions, there are three main approaches to reduce its inference latency: make it do inference faster, make the model smaller, or make the hardware it's deployed on run faster.

The process of making a model smaller is called model compression, and the process to make it do inference faster is called inference optimization. Originally, model compression was to make models fit on edge devices. However, making models smaller often makes them run faster.

We'll discuss inference optimization in the section "Model optimization", and we'll discuss the landscape for hardware backends being developed specifically for running ML models faster in the section "ML on the Cloud and on the Edge". Here, we'll discuss model compression.

The number of research papers on model compression is growing. Off-the-shelf utilities are proliferating. As of April 2022, Awesome Open Source has a list of "The Top 168 Model Compression Open Source Projects", and that list is growing. While there are many new techniques being developed, the four types of techniques that you might come across the most often are low-rank optimization, knowledge distillation, pruning, and quantization. Readers interested in a comprehensive review might want to check out Cheng et al.'s "Survey of Model Compression and Acceleration for Deep Neural Networks," which was updated in 2020.[19]

Low-Rank Factorization

The key idea behind *low-rank factorization* is to replace high-dimensional tensors with lower-dimensional tensors.[20] One type of low-rank factorization is *compact convolutional filters*, where the over-parameterized (having too many parameters) convolution filters are replaced with compact blocks to both reduce the number of parameters and increase speed.

For example, by using a number of strategies including replacing 3 × 3 convolution with 1 × 1 convolution, SqueezeNets achieves AlexNet-level accuracy on ImageNet with 50 times fewer parameters.[21]

Similarly, MobileNets decomposes the standard convolution of size $K \times K \times C$ into a depthwise convolution ($K \times K \times 1$) and a pointwise convolution ($1 \times 1 \times C$), with K being the kernel size and C being the number of channels. This means that each new convolution uses only $K^2 + C$ instead of K^2C parameters. If $K = 3$, this means an eight to nine times reduction in the number of parameters (see Figure 7-9).[22]

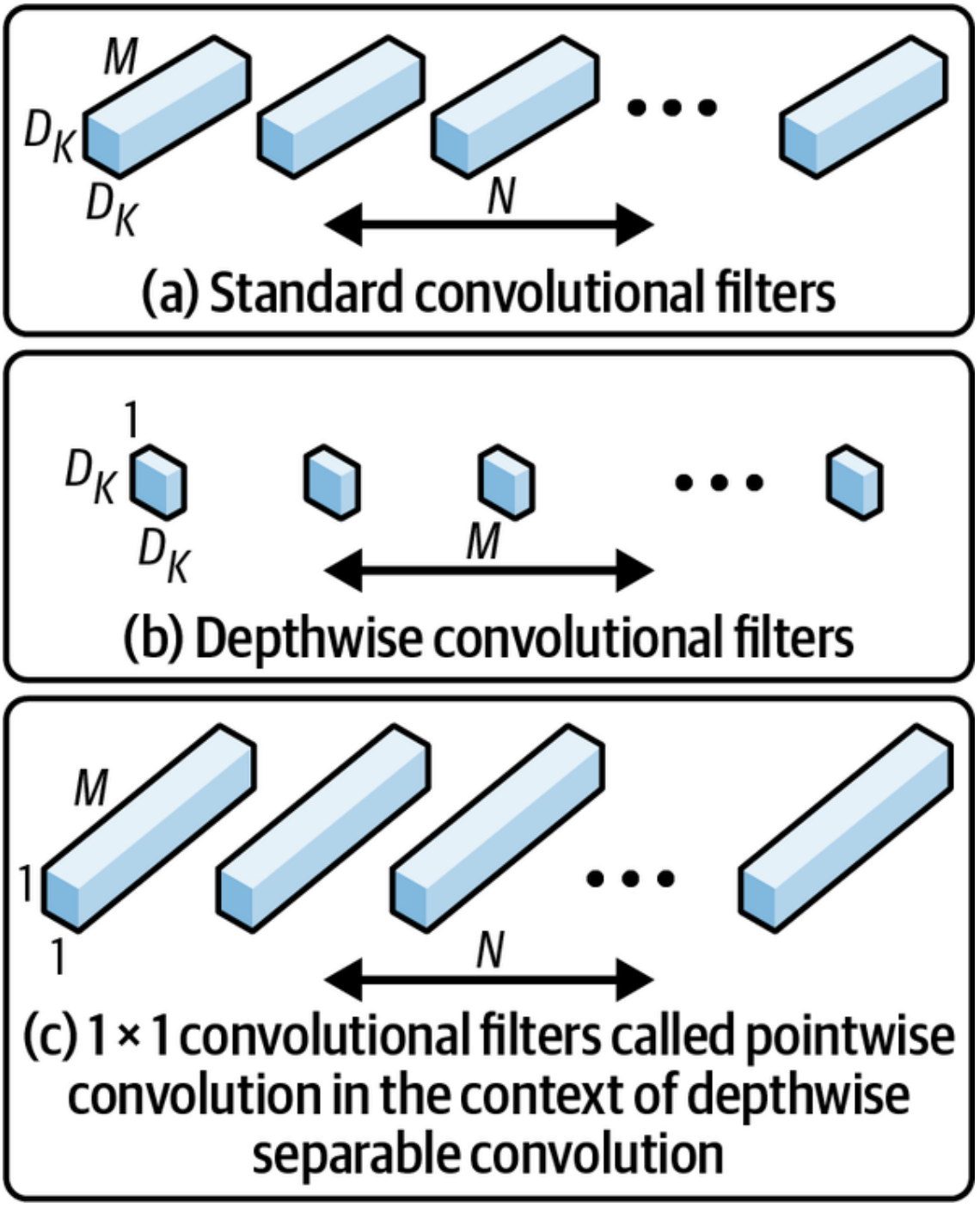

Figure 7-9. Compact convolutional filters in MobileNets. The standard convolutional filters in (a) are replaced by depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter. Source: Adapted from an image by Howard et al.

This method has been used to develop smaller models with significant acceleration compared to standard models. However, it tends to be specific to certain types of models (e.g., compact convolutional filters are specific to convolutional neural networks) and requires a lot of architectural knowledge to design, so it's not widely applicable to many use cases yet.

Knowledge Distillation

Knowledge distillation is a method in which a small model (student) is trained to mimic a larger model or ensemble of models (teacher). The smaller model is what you'll deploy. Even though the student is often trained after a pretrained teacher, both may also be trained at the same time.[23] One example of a distilled network used in production is DistilBERT, which reduces the size of a BERT model by 40% while retaining 97% of its language understanding capabilities and being 60% faster.[24]

The advantage of this approach is that it can work regardless of the architectural differences between the teacher and the student networks. For example, you can get a random forest as the student and a transformer as the teacher. The disadvantage of this approach is that it's highly dependent on the availability of a teacher network. If you use a pretrained model as the teacher model, training the student network will require less data and will likely be faster. However, if you don't have a teacher available, you'll have to train a teacher network before training a student network, and training a teacher network will require a lot more data and take more time to train. This method is also sensitive to applications and model architectures, and therefore hasn't found wide usage in production.

Pruning

Pruning was a method originally used for decision trees where you remove sections of a tree that are uncritical and redundant for classification.[25] As neural networks gained wider adoption, people started to realize that neural networks are over-parameterized and began to find ways to reduce the workload caused by the extra parameters.

Pruning, in the context of neural networks, has two meanings. One is to remove entire nodes of a neural network, which means changing its architecture and reducing its number of parameters. The more common meaning is to find parameters least useful to predictions and set them to 0. In this case, pruning doesn't reduce the total number of parameters, only the number of nonzero parameters. The architecture of the neural network remains the same. This helps with reducing the size of a model

because pruning makes a neural network more sparse, and sparse architecture tends to require less storage space than dense structure. Experiments show that pruning techniques can reduce the nonzero parameter counts of trained networks by over 90%, decreasing storage requirements and improving computational performance of inference without compromising overall accuracy.[26] In Chapter 11, we'll discuss how pruning can introduce biases into your model.

While it's generally agreed that pruning works,[27] there have been many discussions on the actual value of pruning. Liu et al. argued that the main value of pruning isn't in the inherited "important weights" but in the pruned architecture itself.[28] In some cases, pruning can be useful as an architecture search paradigm, and the pruned architecture should be retrained from scratch as a dense model. However, Zhu et al. showed that the large sparse model after pruning outperformed the retrained dense counterpart.[29]

Quantization

Quantization is the most general and commonly used model compression method. It's straightforward to do and generalizes over tasks and architectures.

Quantization reduces a model's size by using fewer bits to represent its parameters. By default, most software packages use 32 bits to represent a float number (single precision floating point). If a model has 100M parameters and each requires 32 bits to store, it'll take up 400 MB. If we use 16 bits to represent a number, we'll reduce the memory footprint by half. Using 16 bits to represent a float is called half precision.

Instead of using floats, you can have a model entirely in integers; each integer takes only 8 bits to represent. This method is also known as "fixed point." In the extreme case, some have attempted the 1-bit representation of each weight (binary weight neural networks), e.g., BinaryConnect and XNOR-Net.[30] The authors of the XNOR-Net paper spun off Xnor.ai, a startup that focused on model compression. In early 2020, it was acquired by Apple for a reported $200M.[31]

Quantization not only reduces memory footprint but also improves the computation speed. First, it allows us to increase our batch size. Second, less precision speeds up computation, which further reduces training time and inference latency. Consider the addition of two numbers. If we perform the addition bit by bit, and each takes *x* nanoseconds, it'll take 32*x* nanoseconds for 32-bit numbers but only 16*x* nanoseconds for 16-bit numbers.

There are downsides to quantization. Reducing the number of bits to represent your numbers means that you can represent a smaller range of values. For values outside that range, you'll have to round them up and/or scale them to be in range. Rounding numbers leads to rounding errors, and small rounding errors can lead to big performance changes. You also run the risk of rounding/scaling your numbers to under-/overflow and rendering it to 0. Efficient rounding and scaling is nontrivial to implement at a low level, but luckily, major frameworks have this built in.

Quantization can either happen during training (quantization aware training),[32] where models are trained in lower precision, or post-training, where models are trained in single-precision floating point and then quantized for inference. Using quantization during training means that you can use less memory for each parameter, which allows you to train larger models on the same hardware.

Recently, low-precision training has become increasingly popular, with support from most modern training hardware. NVIDIA introduced Tensor Cores, processing units that support mixed-precision training.[33] Google TPUs (tensor processing units) also support training with Bfloat16 (16-bit Brain Floating Point Format), which the company dubbed "the secret to high performance on Cloud TPUs."[34] Training in fixed-point is not yet as popular but has had a lot of promising results.[35]

Fixed-point inference has become a standard in the industry. Some edge devices only support fixed-point inference. Most popular frameworks for on-device ML inference—Google's TensorFlow Lite, Facebook's PyTorch Mobile, NVIDIA's TensorRT—offer post-training quantization for free with a few lines of code.

CASE STUDY

To get a better understanding of how to optimize models in production, consider a fascinating case study from Roblox on how they scaled BERT to serve 1+ billion daily requests on CPUs.[36] For many of their NLP services, they needed to handle over 25,000 inferences per second at a latency of under 20 ms, as shown in Figure 7-10. They started with a large BERT model with fixed shape input, then replaced BERT with DistilBERT and fixed shape input with dynamic shape input, and finally quantized it.

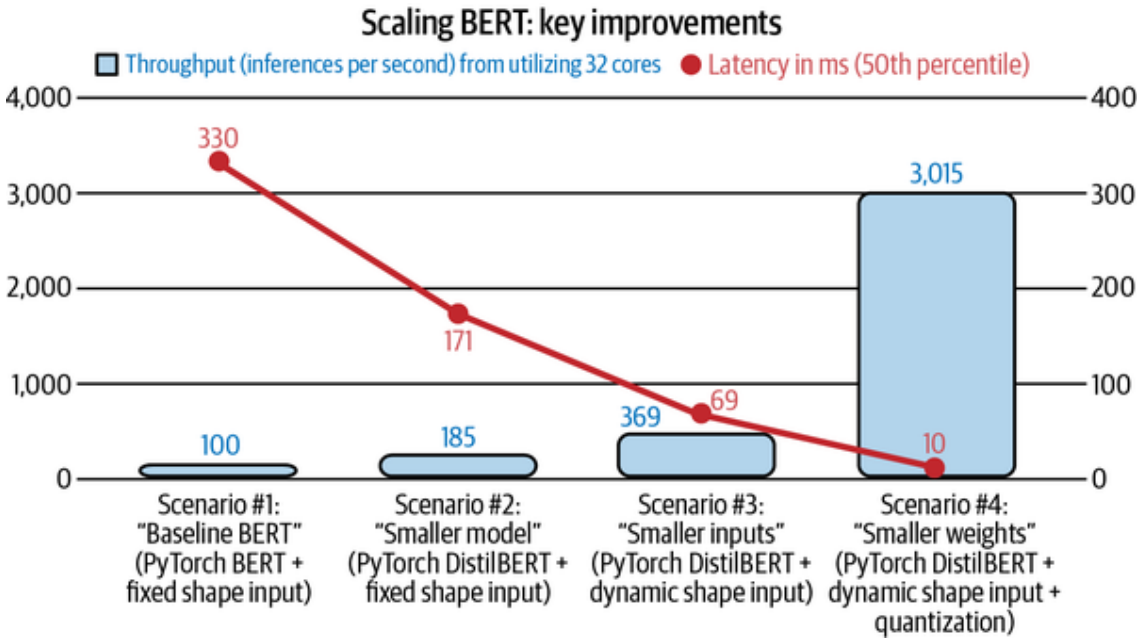

Figure 7-10. Latency improvement by various model compression methods. Source: Adapted from an image by Le and Kaehler

The biggest performance boost they got came from quantization. Converting 32-bit floating points to 8-bit integers reduces the latency 7 times and increases throughput 8 times.

The results here seem very promising to improve latency; however, they should be taken with a grain of salt since there's no mention of changes in output quality after each performance improvement.

ML on the Cloud and on the Edge

Another decision you'll want to consider is where your model's computation will happen: on the cloud or on the edge. On the cloud means a large chunk of computation is done on the cloud, either public clouds or private clouds. On the edge means a large chunk of computation is done on consumer devices—such as browsers, phones, laptops, smartwatches, cars, security cameras, robots, embedded devices, FPGAs (field programmable gate arrays), and ASICs (application-specific integrated circuits)—which are also known as edge devices.

The easiest way is to package your model up and deploy it via a managed cloud service such as AWS or GCP, and this is how many companies deploy when they get started in ML. Cloud services have done an incredible job to make it easy for companies to bring ML models into production.

However, there are many downsides to cloud deployment. The first is cost. ML models can be compute-intensive, and compute is expensive. Even back in 2018, big companies like Pinterest, Infor, and Intuit were already spending hundreds of millions of dollars on cloud bills every year.[37] That number for small and medium companies can be between $50K and $2M a year.[38] A mistake in handling cloud services can cause startups to go bankrupt.[39]

As their cloud bills climb, more and more companies are looking for ways to push their computations to edge devices. The more computation is done on the edge, the less is required on the cloud, and the less they'll have to pay for servers.

Other than help with controlling costs, there are many properties that make edge computing appealing. The first is that it allows your applications to run where cloud computing cannot. When your models are on public clouds, they rely on stable internet connections to send data to the cloud and back. Edge computing allows your models to work in situations where there are no internet connections or where the connections are unreliable, such as in rural areas or developing countries. I've worked with several companies and organizations that have strict no-internet policies, which means that whichever applications we wanted to sell them must not rely on internet connections.

Second, when your models are already on consumers' devices, you can worry less about network latency. Requiring data transfer over the network (sending data to the model on the cloud to make predictions then sending predictions back to the users) might make some use cases impossible. In many cases, network latency is a bigger bottleneck than inference latency. For example, you might be able to reduce the inference latency of ResNet-50 from 30 ms to 20 ms, but the network latency can go up to seconds, depending on where you are and what services you're trying to use.

Putting your models on the edge is also appealing when handling sensitive user data. ML on the cloud means that your systems might have to send user data over networks, making it susceptible to being intercepted. Cloud computing also often means storing data of many users in the same place, which means a breach can affect many people. "Nearly 80% of companies experienced a cloud data breach in [the] past 18 months," according to *Security* magazine.[40]

Edge computing makes it easier to comply with regulations, like GDPR, about how user data can be transferred or stored. While edge computing might reduce privacy concerns, it doesn't eliminate them altogether. In some cases, edge computing might make it easier for attackers to steal user data, such as they can just take the device with them.

To move computation to the edge, the edge devices have to be powerful enough to handle the computation, have enough memory to store ML models and load them into memory, as well as have enough battery or be connected to an energy source to power the application for a reasonable amount of time. Running a full-sized BERT on your phone, if your phone is capable of running BERT, is a very quick way to kill its battery.

Because of the many benefits that edge computing has over cloud computing, companies are in a race to develop edge devices optimized for different ML use cases. Established companies including Google, Apple, and Tesla have all announced their plans to make their own chips. Meanwhile, ML hardware startups have raised billions of dollars to de-

velop better AI chips.[41] It's projected that by 2025 the number of active edge devices worldwide will be over 30 billion.[42]

With so many new offerings for hardware to run ML models on, one question arises: how do we make our model run on arbitrary hardware efficiently? In the following section, we'll discuss how to compile and optimize a model to run it on a certain hardware backend. In the process, we'll introduce important concepts that you might encounter when handling models on the edge, including intermediate representations (IRs) and compilers.

Compiling and Optimizing Models for Edge Devices

For a model built with a certain framework, such as TensorFlow or PyTorch, to run on a hardware backend, that framework has to be supported by the hardware vendor. For example, even though TPUs were released publicly in February 2018, it wasn't until September 2020 that PyTorch was supported on TPUs. Before then, if you wanted to use a TPU, you'd have to use a framework that TPUs supported.

Providing support for a framework on a hardware backend is time-consuming and engineering-intensive. Mapping from ML workloads to a hardware backend requires understanding and taking advantage of that hardware's design, and different hardware backends have different memory layouts and compute primitives, as shown in Figure 7-11.

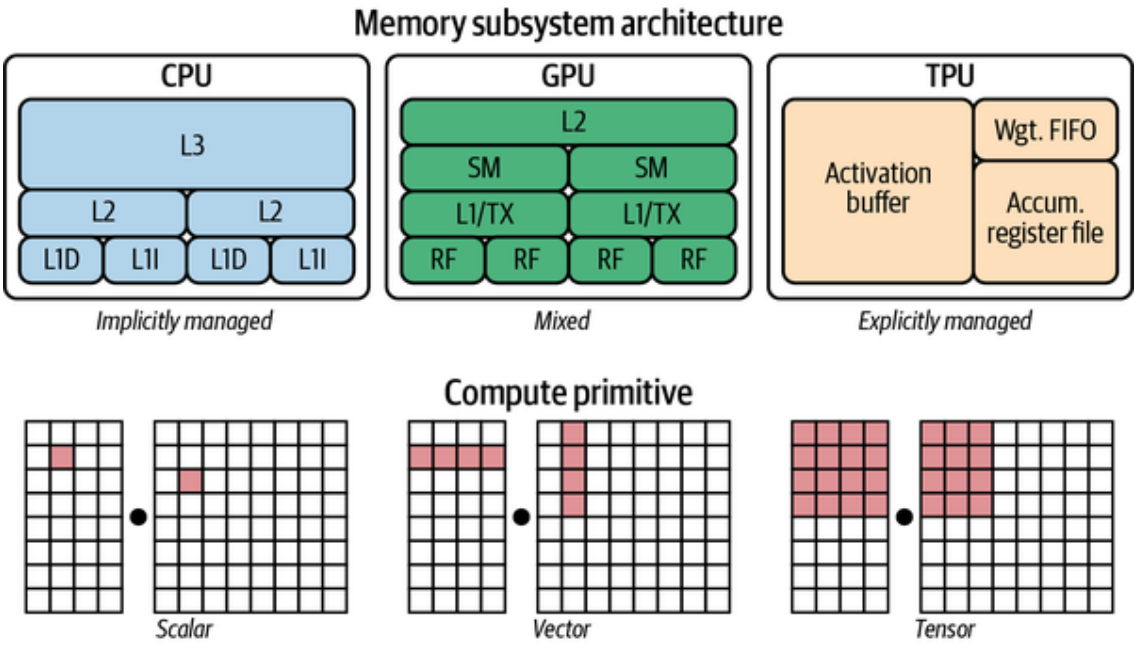

Figure 7-11. Different compute primitives and memory layouts for CPU, GPU, and TPU. Source: Adapted from an image by Chen et al.[43]

For example, the compute primitive of CPUs used to be a number (scalar) and the compute primitive of GPUs used to be a one-dimensional vector, whereas the compute primitive of TPUs is a two-dimensional vector (tensor).[44] Performing a convolution operator will be very different with one-dimensional vectors compared to two-dimensional vectors. Similarly, you'd need to take into account different L1, L2, and L3 layouts and buffer sizes to use them efficiently.

Because of this challenge, framework developers tend to focus on providing support to only a handful of server-class hardware, and hardware vendors tend to offer their own kernel libraries for a narrow range of frameworks. Deploying ML models to new hardware requires significant manual effort.

Instead of targeting new compilers and libraries for every new hardware backend, what if we create a middleman to bridge frameworks and platforms? Framework developers will no longer have to support every type of hardware; they will only need to translate their framework code into this middleman. Hardware vendors can then support one middleman instead of multiple frameworks.

This type of "middleman" is called an intermediate representation (IR). IRs lie at the core of how compilers work. From the original code for a

model, compilers generate a series of high- and low-level IRs before generating the code native to a hardware backend so that it can run on that hardware backend, as shown in Figure 7-12.

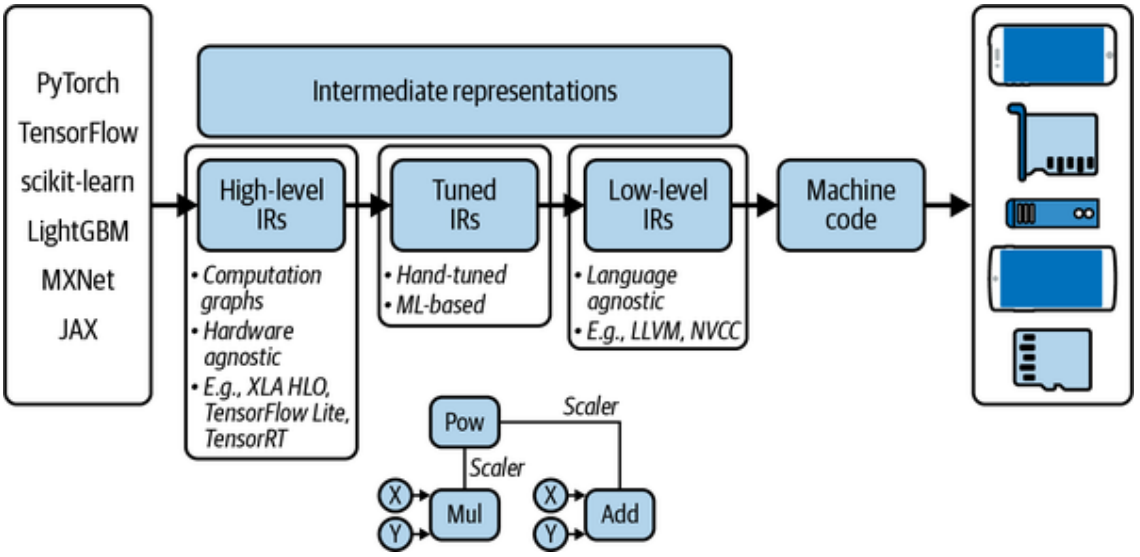

Figure 7-12. A series of high- and low-level IRs between the original model code to machine code that can run on a given hardware backend

This process is also called *lowering*, as in you "lower" your high-level framework code into low-level hardware-native code. It's not translating because there's no one-to-one mapping between them.

High-level IRs are usually computation graphs of your ML models. A computation graph is a graph that describes the order in which your computation is executed. Readers interested can read about computation graphs in PyTorch and TensorFlow.

Model optimization

After you've "lowered" your code to run your models into the hardware of your choice, an issue you might run into is performance. The generated machine code might be able to run on a hardware backend, but it might not be able to do so efficiently. The generated code may not take advantage of data locality and hardware caches, or it may not leverage advanced features such as vector or parallel operations that could speed code up.

A typical ML workflow consists of many frameworks and libraries. For example, you might use pandas/dask/ray to extract features from your

data. You might use NumPy to perform vectorization. You might use a pretrained model like Hugging Face's Transformers to generate features, then make predictions using an ensemble of models built with various frameworks like sklearn, TensorFlow, or LightGBM.

Even though individual functions in these frameworks might be optimized, there's little to no optimization across frameworks. A naive way of moving data across these functions for computation can cause an order of magnitude slowdown in the whole workflow. A study by researchers at Stanford DAWN lab found that typical ML workloads using NumPy, pandas, and TensorFlow run *23 times slower* in one thread compared to hand-optimized code.[45]

In many companies, what usually happens is that data scientists and ML engineers develop models that seem to be working fine in development. However, when these models are deployed, they turn out to be too slow, so their companies hire optimization engineers to optimize their models for the hardware their models run on. An example of a job description for optimization engineers at Mythic follows:

> *This vision comes together in the AI Engineering team, where our expertise is used to develop AI algorithms and models that are optimized for our hardware, as well as to provide guidance to Mythic's hardware and compiler teams.*
>
> *The AI Engineering team significantly impacts Mythic by:*
>
> - *Developing quantization and robustness AI retraining tools*
> - *Investigating new features for our compiler that leverage the adaptability of neural networks*
> - *Developing new neural networks that are optimized for our hardware products*
> - *Interfacing with internal and external customers to meet their development needs*

Optimization engineers are hard to come by and expensive to hire because they need to have expertise in both ML and hardware architectures. Optimizing compilers (compilers that also optimize your code) are

an alternative solution, as they can automate the process of optimizing models. In the process of lowering ML model code into machine code, compilers can look at the computation graph of your ML model and the operators it consists of—convolution, loops, cross-entropy—and find a way to speed it up.

There are two ways to optimize your ML models: locally and globally. Locally is when you optimize an operator or a set of operators of your model. Globally is when you optimize the entire computation graph end to end.

There are standard local optimization techniques that are known to speed up your model, most of them making things run in parallel or reducing memory access on chips. Here are four of the common techniques:

Vectorization

Given a loop or a nested loop, instead of executing it one item at a time, execute multiple elements contiguous in memory at the same time to reduce latency caused by data I/O.

Parallelization

Given an input array (or *n*-dimensional array), divide it into different, independent work chunks, and do the operation on each chunk individually.

Loop tiling[46]

Change the data accessing order in a loop to leverage hardware's memory layout and cache. This kind of optimization is hardware dependent. A good access pattern on CPUs is not a good access pattern on GPUs.

Operator fusion

Fuse multiple operators into one to avoid redundant memory access. For example, two operations on the same array require two loops over that array. In a fused case, it's just one loop. Figure 7-13 shows an example of operator fusion.

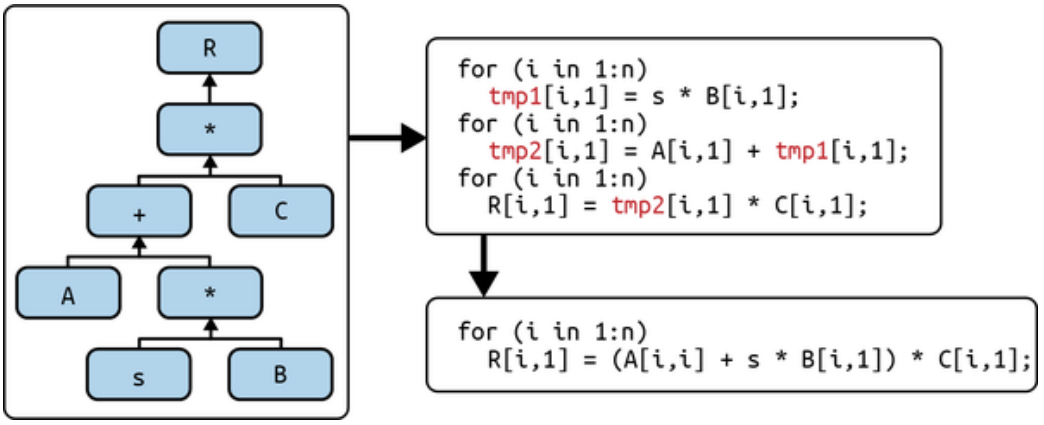

Figure 7-13. An example of an operator fusion. Source: Adapted from an image by Matthias Boehm[47]

To obtain a much bigger speedup, you'd need to leverage higher-level structures of your computation graph. For example, a convolution neural network with the computation graph can be fused vertically or horizontally to reduce memory access and speed up the model, as shown in Figure 7-14.

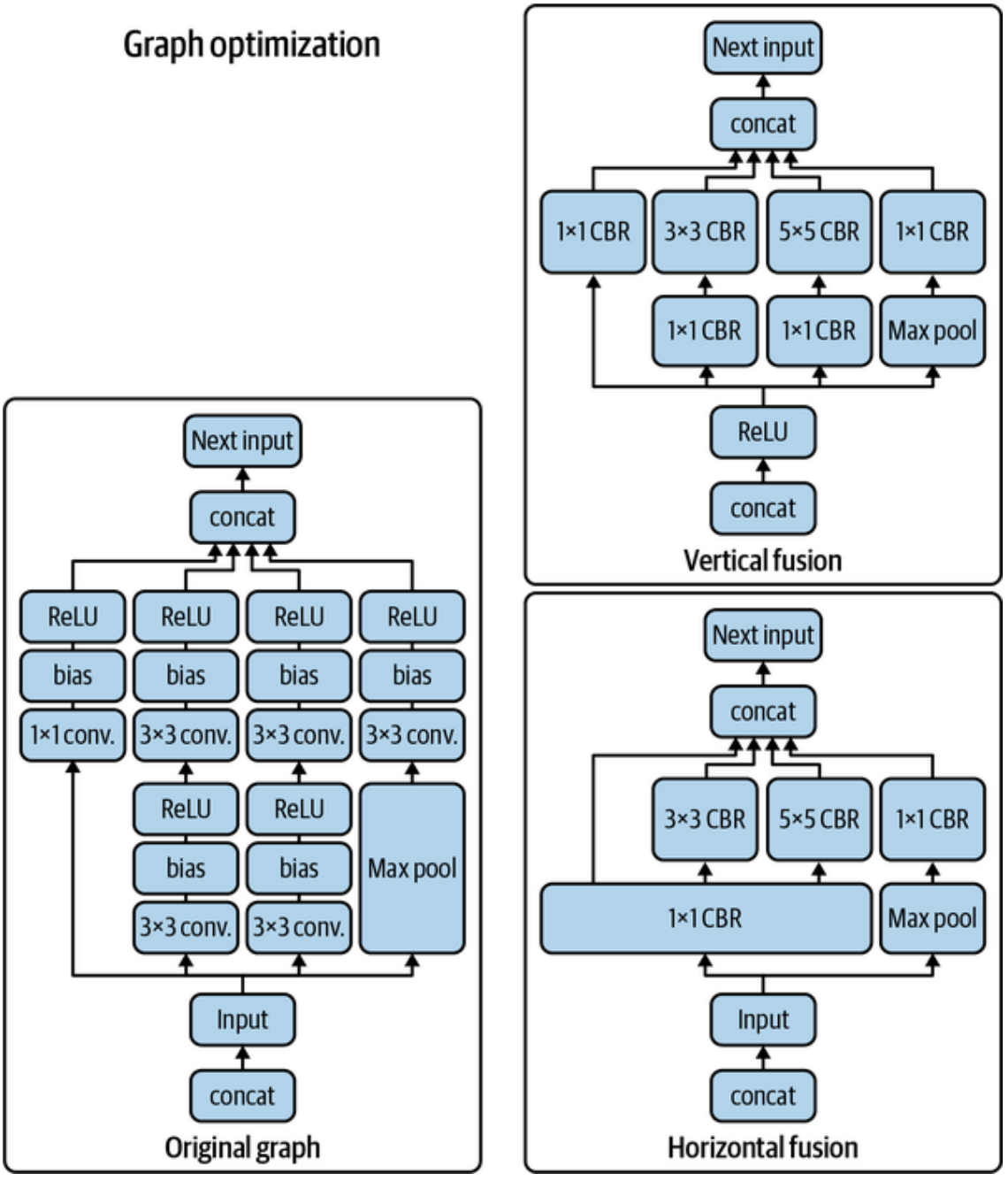

Figure 7-14. Vertical and horizontal fusion of the computation graph of a convolution neural network. Source: Adapted from an image by TensorRT team[48]

Using ML to optimize ML models

As hinted by the previous section with the vertical and horizontal fusion for a convolutional neural network, there are many possible ways to execute a given computation graph. For example, given three operators A, B, and C, you can fuse A with B, fuse B with C, or fuse A, B, and C all together.

Traditionally, framework and hardware vendors hire optimization engineers who, based on their experience, come up with heuristics on how to best execute the computation graph of a model. For example, NVIDIA

might have an engineer or a team of engineers who focus exclusively on how to make ResNet-50 run really fast on their DGX A100 server.[49]

There are a couple of drawbacks to hand-designed heuristics. First, they're nonoptimal. There's no guarantee that the heuristics an engineer comes up with are the best possible solution. Second, they are nonadaptive. Repeating the process on a new framework or a new hardware architecture requires an enormous amount of effort.

This is complicated by the fact that model optimization is dependent on the operators its computation graph consists of. Optimizing a convolution neural network is different from optimizing a recurrent neural network, which is different from optimizing a transformer. Hardware vendors like NVIDIA and Google focus on optimizing popular models like ResNet-50 and BERT for their hardware. But what if you, as an ML researcher, come up with a new model architecture? You might need to optimize it yourself to show that it's fast first before it's adopted and optimized by hardware vendors.

If you don't have ideas for good heuristics, one possible solution might be to try all possible ways to execute a computation graph, record the time they need to run, then pick the best one. However, given a combinatorial number of possible paths, exploring them all would be intractable. Luckily, approximating the solutions to intractable problems is what ML is good at. What if we use ML to narrow down the search space so we don't have to explore that many paths, and predict how long a path will take so that we don't have to wait for the entire computation graph to finish executing?

To estimate how much time a path through a computation graph will take to run turns out to be difficult, as it requires making a lot of assumptions about that graph. It's much easier to focus on a small part of the graph.

If you use PyTorch on GPUs, you might have seen `torch.backends.cudnn.benchmark=True`. When this is set to True, *cuDNN autotune* will be enabled. cuDNN autotune searches over a predetermined set of options to execute a convolution operator and then chooses the fastest way. cuDNN autotune, despite its effectiveness, only

works for convolution operators. A much more general solution is autoTVM, which is part of the open source compiler stack TVM. autoTVM works with subgraphs instead of just an operator, so the search spaces it works with are much more complex. The way autoTVM works is quite complicated, but in simple terms:

1. It first breaks your computation graph into subgraphs.
2. It predicts how big each subgraph is.
3. It allocates time to search for the best possible path for each subgraph.
4. It stitches the best possible way to run each subgraph together to execute the entire graph.

autoTVM measures the actual time it takes to run each path it goes down, which gives it ground truth data to train a cost model to predict how long a future path will take. The pro of this approach is that because the model is trained using the data generated during runtime, it can adapt to any type of hardware it runs on. The con is that it takes more time for the cost model to start improving. Figure 7-15 shows the performance gain that autoTVM gave compared to cuDNN for the model ResNet-50 on NVIDIA TITAN X.

While the results of ML-powered compilers are impressive, they come with a catch: they can be slow. You go through all the possible paths and find the most optimized ones. This process can take hours, even days for complex ML models. However, it's a one-time operation, and the results of your optimization search can be cached and used to both optimize existing models and provide a starting point for future tuning sessions. You optimize your model once for one hardware backend then run it on multiple devices of that same hardware type. This sort of optimization is ideal when you have a model ready for production and target hardware to run inference on.

Figure 7-15. Speedup achieved by autoTVM over cuDNN for ResNet-50 on NVIDIA TITAN X. It takes ~70 trials for autoTVM to outperform cuDNN. Source: Chen et al.[50]

ML in Browsers

We've been talking about how compilers can help us generate machine-native code run models on certain hardware backends. It is, however, possible to generate code that can run on just any hardware backends by running that code in browsers. If you can run your model in a browser, you can run your model on any device that supports browsers: MacBooks, Chromebooks, iPhones, Android phones, and more. You wouldn't need to care what chips those devices use. If Apple decides to switch from Intel chips to ARM chips, it's not your problem.

When talking about browsers, many people think of JavaScript. There are tools that can help you compile your models into JavaScript, such as TensorFlow.js, Synaptic, and brain.js. However, JavaScript is slow, and its capacity as a programming language is limited for complex logics such as extracting features from data.

A more promising approach is WebAssembly (WASM). WASM is an open standard that allows you to run executable programs in browsers. After you've built your models in scikit-learn, PyTorch, TensorFlow, or whatever frameworks you've used, instead of compiling your models to run on specific hardware, you can compile your model to WASM. You get back an executable file that you can just use with JavaScript.

WASM is one of the most exciting technological trends I've seen in the last couple of years. It's performant, easy to use, and has an ecosystem that is growing like wildfire.[51] As of September 2021, it's supported by 93% of devices worldwide.[52]

The main drawback of WASM is that because WASM runs in browsers, it's slow. Even though WASM is already much faster than JavaScript, it's still slow compared to running code natively on devices (such as iOS or Android apps). A study by Jangda et al. showed that applications compiled to WASM run slower than native applications by an average of 45% (on Firefox) to 55% (on Chrome).[53]

Summary

Congratulations, you've finished possibly one of the most technical chapters in this book! The chapter is technical because deploying ML models is an engineering challenge, not an ML challenge.

We've discussed different ways to deploy a model, comparing online prediction with batch prediction, and ML on the edge with ML on the cloud. Each way has its own challenges. Online prediction makes your model more responsive to users' changing preferences, but you have to worry about inference latency. Batch prediction is a workaround for when your models take too long to generate predictions, but it makes your model less flexible.

Similarly, doing inference on the cloud is easy to set up, but it becomes impractical with network latency and cloud cost. Doing inference on the edge requires having edge devices with sufficient compute power, memory, and battery.

However, I believe that most of these challenges are due to the limitations of the hardware that ML models run on. As hardware becomes more powerful and optimized for ML, I believe that ML systems will transition to making online prediction on-device, illustrated in Figure 7-16.

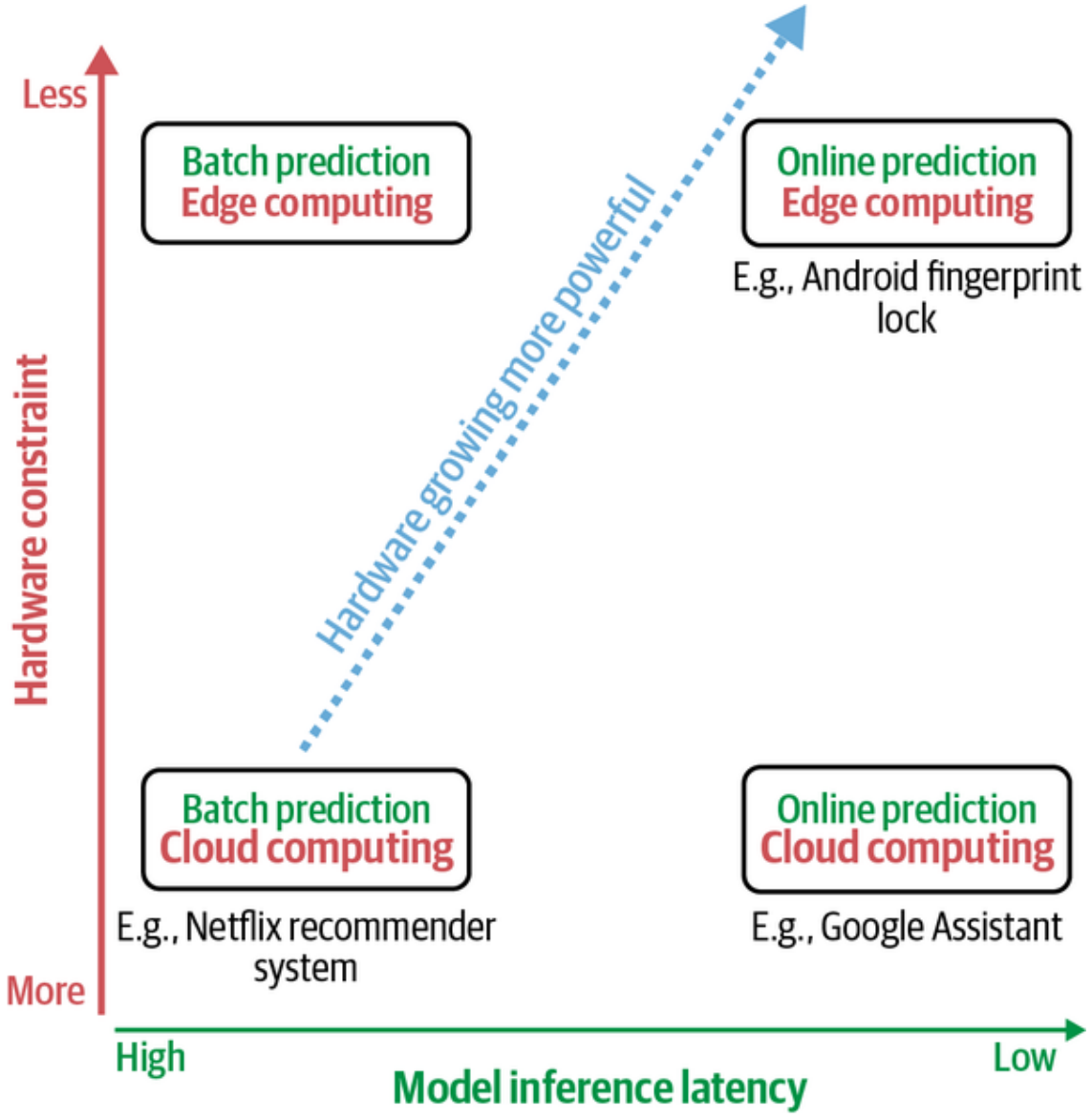

Figure 7-16. As hardware becomes more powerful, ML models will move to online and on the edge

I used to think that an ML project is done after the model is deployed, and I hope that I've made clear in this chapter that I was seriously mistaken. Moving the model from the development environment to the production environment creates a whole new host of problems. The first is how to keep that model in production. In the next chapter, we'll discuss how our models might fail in production, and how to continually monitor models to detect issues and address them as fast as possible.

1 We'll cover development environments in detail in Chapter 10.

2 We'll go more into containers in Chapter 9.

3 CS 329S: Machine Learning Systems Design at Stanford; you can see the project demos on YouTube.

4 See the discussion on "data serialization" in the section "Data Formats".

5 Ville Tuulos, "Human-Centric Machine Learning Infrastructure @Netflix," InfoQ, 2018, video, 49:11, *https://oreil.ly/j4Hfx*.

6 Wayne Cunningham, "Science at Uber: Powering Machine Learning at Uber," *Uber Engineering Blog*, September 10, 2019, *https://oreil.ly/WfaCF*.

7 Daniel Papasian and Todd Underwood, "OpML '20—How ML Breaks: A Decade of Outages for One Large ML Pipeline," Google, 2020, video, 19:06, *https://oreil.ly/HjQm0*.

8 Lucas Bernardi, Themistoklis Mavridis, and Pablo Estevez, "150 Successful Machine Learning Models: 6 Lessons Learned at Booking.com," *KDD '19: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (July 2019): 1743–51, *https://oreil.ly/Ea1Ke*.

9 "2021 Enterprise Trends in Machine Learning," Algorithmia, *https://oreil.ly/9kdcw*.

10 We'll discuss data distribution shifts further in Chapter 8.

11 Christopher Null, "10 Companies Killing It at DevOps," *TechBeacon*, 2015, *https://oreil.ly/JvNwu*.

12 Qian Yu, "Machine Learning with Flink in Weibo," QCon 2019, video, 17:57, *https://oreil.ly/RcTMv*.

13 Josh Wills, "Instrumentation, Observability and Monitoring of Machine Learning Models," InfoQ 2019, *https://oreil.ly/5Ot5m*.

14 "Developer Survey Results," Stack Overflow, 2019, *https://oreil.ly/guYIq*.

15 David Curry, "Grubhub Revenue and Usage Statistics (2022)," Business of Apps, January 11, 2022, *https://oreil.ly/jX43M*; "Average Number of Grubhub Orders per Day Worldwide from 2011 to 2020," Statista, *https://oreil.ly/Tu9fm*.

16 The URL of the entry point for a service, which, in this case, is the prediction service of your ML model.

17 If a new user joins, you can give them some generic recommendations.

18 Shuyi Chean and Fabian Hueske, “Streaming SQL to Unify Batch & Stream Processing w/ Apache Flink @Uber,” *InfoQ*, https://oreil.ly/XoaNu; Yu, “Machine Learning with Flink in Weibo.”

19 Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang, “A Survey of Model Compression and Acceleration for Deep Neural Networks,” *arXiv*, June 14, 2020, https://oreil.ly/1eMho.

20 Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman, “Speeding up Convolutional Neural Networks with Low Rank Expansions,” *arXiv*, May 15, 2014, https://oreil.ly/4Vf4s.

21 Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer, “SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size,” *arXiv*, November 4, 2016, https://oreil.ly/xs3mi.

22 Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *arXiv*, April 17, 2017, https://oreil.ly/T84fD.

23 Geoffrey Hinton, Oriol Vinyals, and Jeff Dean, “Distilling the Knowledge in a Neural Network,” *arXiv*, March 9, 2015, https://oreil.ly/OJEPW.

24 Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf, “DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter,” *arXiv*, October 2, 2019, https://oreil.ly/mQWBv.

25 Hence the name “pruning.”

26 Jonathan Frankle and Michael Carbin, “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks,” ICLR 2019, https://oreil.ly/ychdl.

27 Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag, “What Is the State of Neural Network Pruning?” *arXiv*, March 6, 2020, https://oreil.ly/VQsC3.

28 Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell, “Rethinking the Value of Network Pruning,” *arXiv*, March 5, 2019, https://oreil.ly/mB4IZ.

29 Michael Zhu and Suyog Gupta, "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression," *arXiv*, November 13, 2017, https://oreil.ly/KBRjy.

30 Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," *arXiv*, November 2, 2015, https://oreil.ly/Fwp2G; Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *arXiv*, August 2, 2016, https://oreil.ly/gr3Ay.

31 Alan Boyle, Taylor Soper, and Todd Bishop, "Exclusive: Apple Acquires Xnor.ai, Edge AI Spin-out from Paul Allen's AI2, for Price in $200M Range," *GeekWire*, January 15, 2020, https://oreil.ly/HgaxC.

32 As of October 2020, TensorFlow's quantization aware training doesn't actually train models with weights in lower bits, but collects statistics to use for post-training quantization.

33 Chip Huyen, Igor Gitman, Oleksii Kuchaiev, Boris Ginsburg, Vitaly Lavrukhin, Jason Li, Vahid Noroozi, and Ravi Gadde, "Mixed Precision Training for NLP and Speech Recognition with OpenSeq2Seq," *NVIDIA Devblogs*, October 9, 2018, https://oreil.ly/WDT1l. It's my post!

34 Shibo Wang and Pankaj Kanwar, "BFloat16: The Secret to High Performance on Cloud TPUs," *Google Cloud Blog*, August 23, 2019, https://oreil.ly/ZG5p0.

35 Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research* 18 (2018): 1–30; Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *arXiv*, December 15, 2017, https://oreil.ly/sUuMT.

36 Quoc Le and Kip Kaehler, "How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs," Roblox, May 27, 2020, https://oreil.ly/U01Uj.

37 Amir Efrati and Kevin McLaughlin, "As AWS Use Soars, Companies Surprised by Cloud Bills," *The Information*, February 25, 2019, https://oreil.ly/H9ans; Mats

Bauer, “How Much Does Netflix Pay Amazon Web Services Each Month?” Quora, 2020, *https://oreil.ly/HtrBk*.

38 “2021 State of Cloud Cost Report,” Anodot, *https://oreil.ly/5ZIJK*.

39 “Burnt $72K Testing Firebase and Cloud Run and Almost Went Bankrupt,” Hacker News, December 10, 2020, *https://oreil.ly/vsHHC*; “How to Burn the Most Money with a Single Click in Azure,” Hacker News, March 29, 2020, *https://oreil.ly/QvCiI*. We’ll discuss in more detail how companies respond to high cloud bills in the section “Public Cloud Versus Private Data Centers”.

40 “Nearly 80% of Companies Experienced a Cloud Data Breach in Past 18 Months,” *Security*, June 5, 2020, *https://oreil.ly/gA1am*.

41 See slide #53, CS 329S’s Lecture 8: Deployment - Prediction Service, 2022, *https://oreil.ly/cXTou*.

42 “Internet of Things (IoT) and Non-IoT Active Device Connections Worldwide from 2010 to 2025,” Statista, https://oreil.ly/BChLN.

43 Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, et al., “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning,” *arXiv*, February 12, 2018, *https://oreil.ly/vGnkW*.

44 Nowadays, many CPUs have vector instructions and some GPUs have tensor cores, which are two-dimensional.

45 Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, et al., “Evaluating End-to-End Optimization for Data Analytics Applications in Weld,” *Proceedings of the VLDB Endowment* 11, no. 9 (2018): 1002–15, *https://oreil.ly/ErUIo*.

46 For a helpful visualization of loop tiling, see slide 33 from Colfax Research’s presentation “Access to Caches and Memory”, session 10 of their Programming and Optimization for Intel Architecture: Hands-on Workshop series. The entire series is available at *https://oreil.ly/hT1g4*.

47 Matthias Boehm, “Architecture of ML Systems 04 Operator Fusion and Runtime Adaptation,” Graz University of Technology, April 5, 2019, *https://oreil.ly/py43J*.

48 Shashank Prasanna, Prethvi Kashinkunti, and Fausto Milletari, “TensorRT 3: Faster TensorFlow Inference and Volta Support,” NVIDIA Developer, December 4,

2017, *https://oreil.ly/d9h98*. CBR stands for “convolution, bias, and ReLU.”

49 This is also why you shouldn’t read too much into benchmarking results, such as MLPerf’s results. A popular model running really fast on a type of hardware doesn’t mean an arbitrary model will run really fast on that hardware. It might just be that this model is over-optimized.

50 Chen et al., “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.”

51 Wasmer, *https://oreil.ly/dTRxr*; Awesome Wasm, *https://oreil.ly/hlIFb*.

52 Can I Use _____?, *https://oreil.ly/slI05*.

53 Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha, “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code,” USENIX, *https://oreil.ly/uVzrX*.