Lecture 8. Model Deployment

Note: *This note is a work-in-progress, created for the course CS 329S: Machine Learning Systems Design (Stanford, 2022). For the fully developed text, see the book Designing Machine Learning Systems (Chip Huyen, O'Reilly 2022).*

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

In this lecture, 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. You can deploy your model on a test environment for testing. You can also deploy it to a production environment to be accessible and usable by your end users. In this lecture, we focus on deploying models to production environments.

However, keep in mind that production is a spectrum. For some teams, production means generating nice plots from notebook results 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.

A wise person on the Internet once said: 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 create an endpoint to your prediction function, push your model to AWS, create an app with Streamlit or Dash. If you're familiar with these tools, you can have a functional deployment in an hour. My students, after a 10-week course[1], were all able to deploy an ML application as their final projects even though few have had deployment experience before.

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 went 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'll be **exported** and handed off from the data scientists or ML engineers 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 9.

[NOTE]
Exporting a model means converting this model into a format that can be used by another application. Some people call this process serialization[2]. 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, how many units in each

[1] CS 329S: Machine Learning Systems Design at Stanford.
[2] See "data serialization" in the Data Formats section in Chapter 2.

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.
[/NOTE]

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.

We'll 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 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. I'll try to take it slow. 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.

[SIDEBAR]
Caution against categorical thinking

The way we categorize a system into batch prediction vs. online prediction and edge computing vs. cloud computing provides an anchor to think about the requirements and limitations of different systems. It's not a guide. Seemingly different ways of doing things might be fundamentally similar, and the choices don't have to be mutually exclusive. For example, you don't have to do only batch predictions or only online predictions -- you can do both. If you're doing batch predictions, switching to online predictions might be easier than you think.

Putting things in buckets might cause organizations to get stuck in one way of doing things without realizing that the other way isn't that different but can provide much more value.
[/SIDEBAR]

Batch Prediction vs. 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.

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 **synchronous prediction**: predictions are generated in synchronization with requests, or **on-demand prediction**: predictions are generated after requests for these predictions, not before. Traditionally, when doing online prediction, requests are sent to the prediction service via RESTful APIs (e.g. HTTP requests — see **Data passing through services** in Chapter 2).

Batch prediction is when predictions are generated periodically or whenever triggered. The predictions are stored somewhere, such as in SQL tables or CSV files, 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 onto Netflix. Batch prediction is also known as **asynchronous prediction**: predictions are generated asynchronously with requests arrive

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**.

Figure 6-4 shows a simplified architecture for batch prediction and Figure 6-5 shows a simplified version of online prediction using only batch features. We'll go over what it means to use only batch features next.

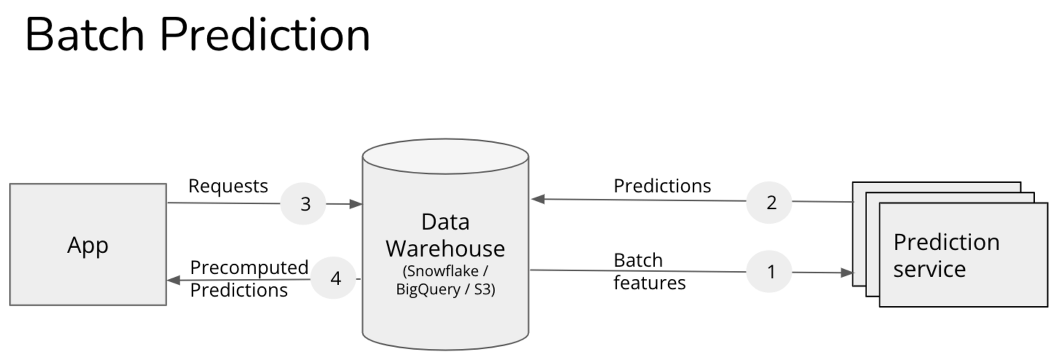


Figure 6-4: A simplified architecture for batch prediction.

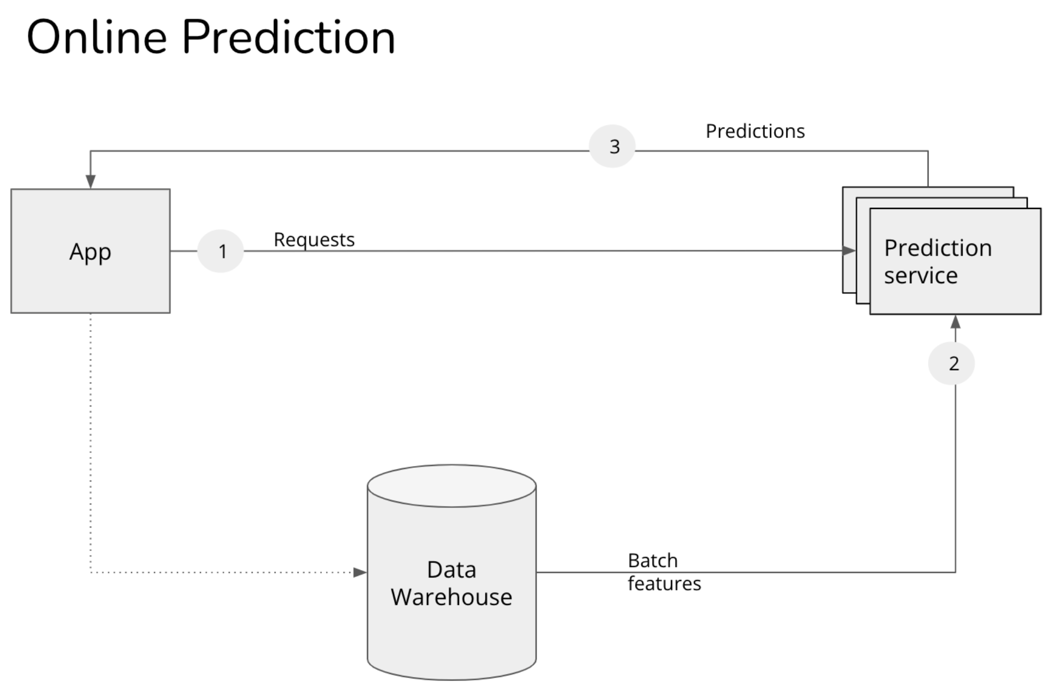


Figure 6-5: A simplified architecture for online prediction that uses only batch features.

As discussed in Chapter 2, features computed from historical data, such as data in databases and data warehouses, are **batch features**. Features computed of 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 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: at this moment, how many other orders they have, how many delivery people are available.

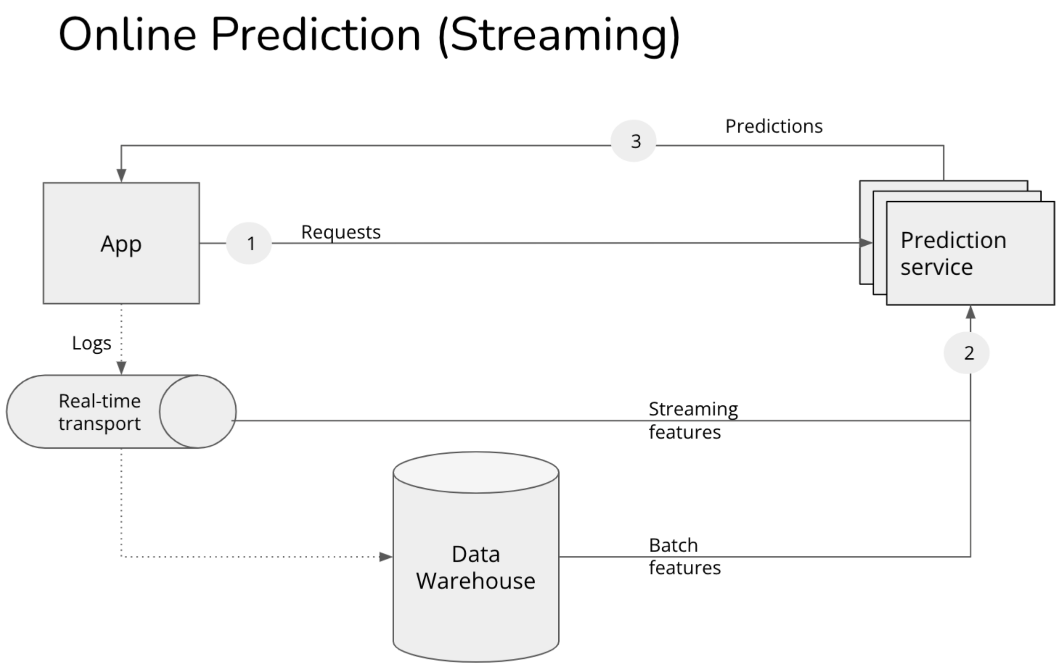


Figure 6-6: A simplified architecture for online prediction that uses both batch features and streaming features.

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

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 6-1 summarizes the key points to consider for online prediction and batch prediction.

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.

	Batch prediction (asynchronous)	**Online prediction (synchronous)**
Frequency	Periodical, such as every 4 hours	As soon as requests come

Useful for	Processing accumulated data when you don't need immediate results (such as recommendation systems)	When predictions are needed as soon as data sample is generated (such as fraud detection)
Optimized for	High throughput	Low latency
Examples	Netflix recommendations	Google Assistant speech recognition

Table 6-1: Some key differences between batch prediction and 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 **Batch Processing vs. Stream Processing** section in Chapter 2.

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. 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 AWS Sagemaker or Google App Engine, and get back an endpoint[3]. Now, if you send a request which contains an input to that endpoint, it will send back a prediction based on that input.

A problem with online prediction is that your model might take too long to generate predictions. If your model takes a couple of seconds too long, your users might get bored.

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? 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

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

reduce the inference latency of more complex models if the time it takes to retrieve a prediction is 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. One of its common use cases is recommendation systems — generating recommendations for users every few hours and only pulling out the recommendation for each user when they log into the system. 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 into Netflix, horror movies dominate recommendations. But you're feeling 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? But it can't update the list until the next batch of recommendations is generated.

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[4]. 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 phones using face or fingerprints, fall detection for elderly care, and fraud detection. Being able to detect a fraudulent transaction that happened 3 hours ago is still better than not detecting it at all, but being able to detect it in real-time can prevent it from going through.

Batch prediction is a workaround for when online prediction isn't cheap enough or isn't fast enough. Why generate 1 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/powerful and better techniques are being developed to allow faster, cheaper online predictions, online prediction might become the default.

[4] If a new user joins, you can give them some generic recommendations.

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 fast-inference model that can generate predictions at a speed acceptable to its end users. For most consumer apps, this means milliseconds.

We'll continue discussing the streaming pipeline and its unification with the batch pipeline in the next section. Then we'll discuss fast inference in the section **Inference 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 machine learning, 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.

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 6-7.

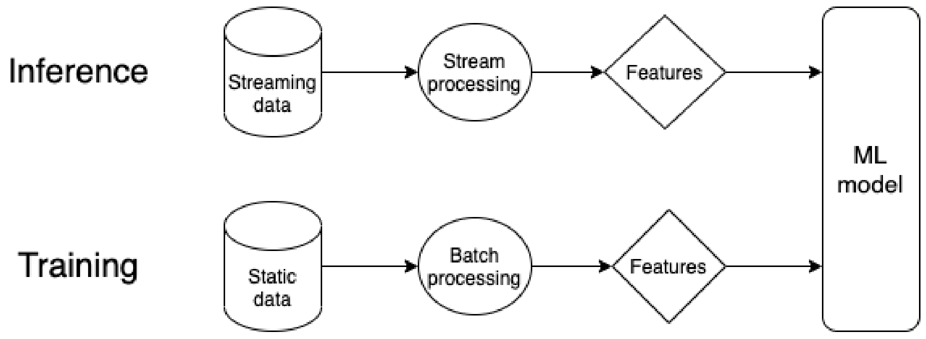


Figure 6-7: Having two different pipelines for training and inference is a common source for bugs for ML in production

Figure 6-8 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.

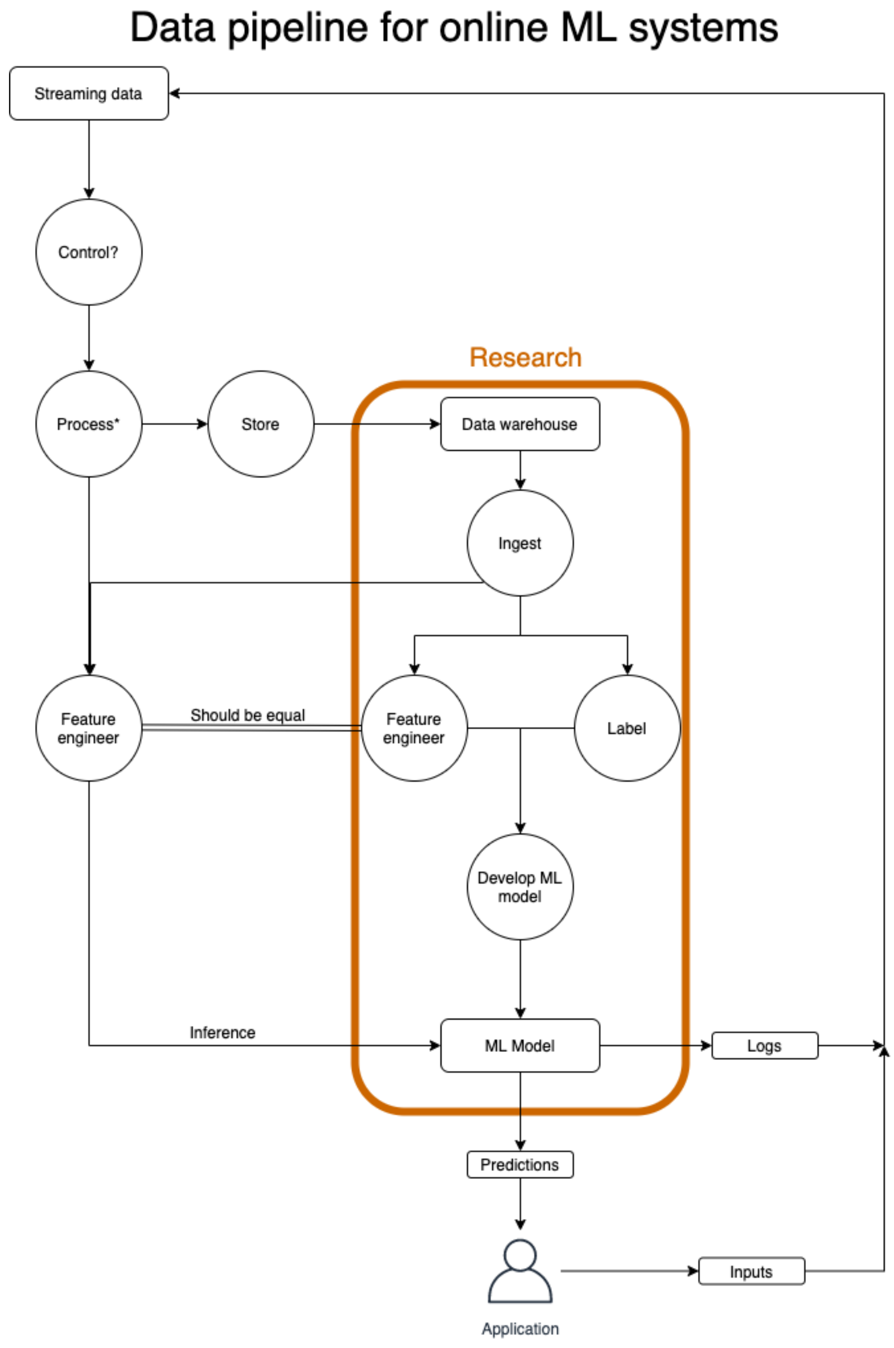


Figure 6-8: A data pipeline for ML systems that do online prediction

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 with Apache Flink[5][6].

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 Edge** later in this chapter. Here, we'll discuss model compression.

The number of research papers on model compression is growing. Off-the-shelf utilities are proliferating. As of September 2021, Awesome Open Source has a list of The Top 108 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 A survey of model compression and acceleration for deep neural networks which was updated in 2020.

Low-rank Factorization

The key idea behind **low-rank factorization**[7] is to replace high-dimensional tensors with lower dimensional tensors. 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.

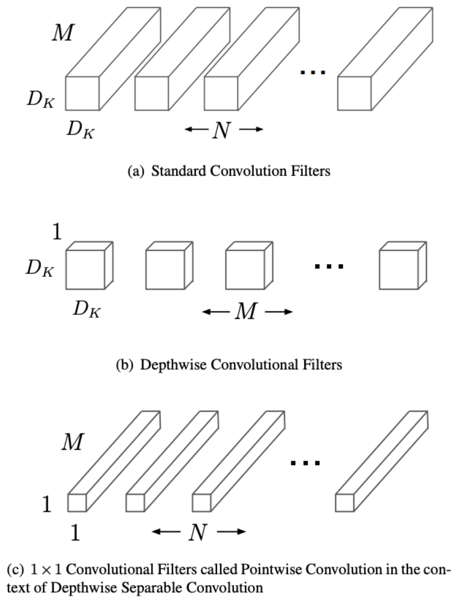
[5] Streaming SQL to Unify Batch & Stream Processing w/ Apache Flink @Uber (InfoQ)

[6] Machine learning with Flink in Weibo - Qian Yu

[7] Speeding up Convolutional Neural Networks with Low Rank Expansions, Jaderberg et al.. 2014.

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[8].

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 a 8x to 9x reduction in the number of parameters[9] (see Figure 6-9).



M
D_K
D_K
$\leftarrow N \rightarrow$
(a) Standard Convolution Filters
1
D_K
D_K
$\leftarrow M \rightarrow$
(b) Depthwise Convolutional Filters
M
1
1
$\leftarrow N \rightarrow$
(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 6-9: 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.
Image by Howard et al.
[TODO: WAITING PERMISSION]

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 (for example, 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.

[8] SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, Iandola. 2016.
[9] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, Howard et al., 2017.

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 pre-trained teacher, both may also be trained at the same time[10]. 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[11].

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 pre-trained 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[12]. As neural networks gain wider adoption, people started to realize that neural networks are over-parameterized, and think about how 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 non-zero 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 non-zero parameter counts of trained networks by over 90%, decreasing storage requirements and improving computational performance of inference without compromising accuracy[13].

---

[10] Distilling the Knowledge in a Neural Network (Hinton et al., 2015)
[11] DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter (Sanh et al., 2019)
[12] Hence the name "pruning", as in "pruning trees".
[13] The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (Frankle et al., ICLR 2019)

While it's generally agreed that pruning works[14], 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. 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 small counterpart.

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, each requires 32 bits to store, it'll take up 400MB. 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[15] and Xnor-Net[16]. 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.

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, each takes x nanosecond, it'll take 32x nanoseconds for 32-bit numbers but only 16x 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-/over-flow and rendering it to 0. Efficient rounding and scaling is non-trivial to implement at a low level, but luckily, major frameworks have this built in.

[14] What is the State of Neural Network Pruning? (Blalock et al., 2020)
[15] Binaryconnect: Training deep neural networks with binary weights during propagations, Courbariaux et al., 2015.
[16] Xnor-net: Imagenet classification using binary convolutional neural networks, Rastegari et al., 2016.

Quantization can either happen during training, which is referred to as quantization aware training[17], where models are trained in lower precision, or post-training, where models are trained in single-precision floating point, then trained models are 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.

In the last few years, 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[18]. Google TPUs also support training with Bfloat16 (16-bit Brain Floating Point Format), which the company dubbed as "*the secret to high performance on Cloud TPUs*"[19]. Training in fixed-point is not yet as popular, but has had a lot of promising results[20][21].

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.

[SIDEBAR]
Case study
To get a better understanding of how to optimize models in production, here's a fascinating case study from Robolox on how they scaled BERT to serve 1+ Billion Daily Requests on CPUs. For many of their NLP services, they needed to handle over 25,000 inferences per second at a latency of under 20ms, as shown in Figure 6-10. 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.

[17] As of October 2020, TensorFlow's quantization aware training doesn't actually train models with weights in lower bits, but collect statistics to use for post-training quantization.
[18] Mixed Precision Training for NLP and Speech Recognition with OpenSeq2Seq (Nguyen et al., NVIDIA Devblogs 2018). It's my post!
[19] BFloat16: The secret to high performance on Cloud TPUs (Pankaj Kanwar, 2019). Google Cloud Blog.
[20] Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations (Hubara et al., 2016)
[21] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., 2017).

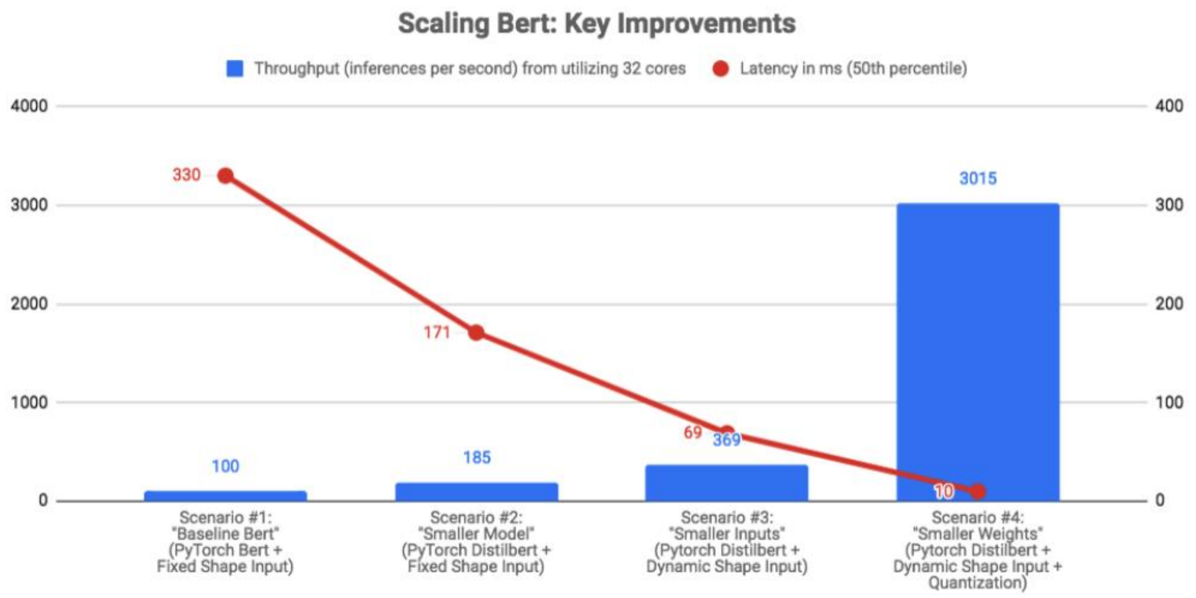


Figure 6-10: Latency improvement by various model compression methods. Experiment done by Quoc Le and Kip Kaehler at Roblox. [Permission granted]

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

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.
[/SIDEBAR]

Earlier in the chapter, we mentioned that to enable online prediction for an ML system, two components are required: a model with fast inference and a (near) real-time pipeline. We've discussed various methods to optimize a model's inference speed. In the next part, we'll discuss a new paradigm that has quickly gained traction in the last five years: stream processing, and how it enables us to build ML systems that can respond in real-time and near real-time. To understand stream processing, we'll contrast it to the older paradigm: batch processing. Let's dive in!

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 the consumer devices — such as browsers, phones, laptops, smartwatches, cars, security cameras, robots, embedded devices, FPGAs[22], and ASICs[23] — 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, Intuit, etc. were already spending hundreds of millions of dollars in cloud bills every year [1, 2]. That number for small and medium companies can be between $50K - 2M a year. A mistake in handling cloud services can cause startups to go bankrupt [1, 2].

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 ResNet50 from 30ms to 20ms, but the network latency can go up to seconds, depending on where you are and what services you're trying to use.

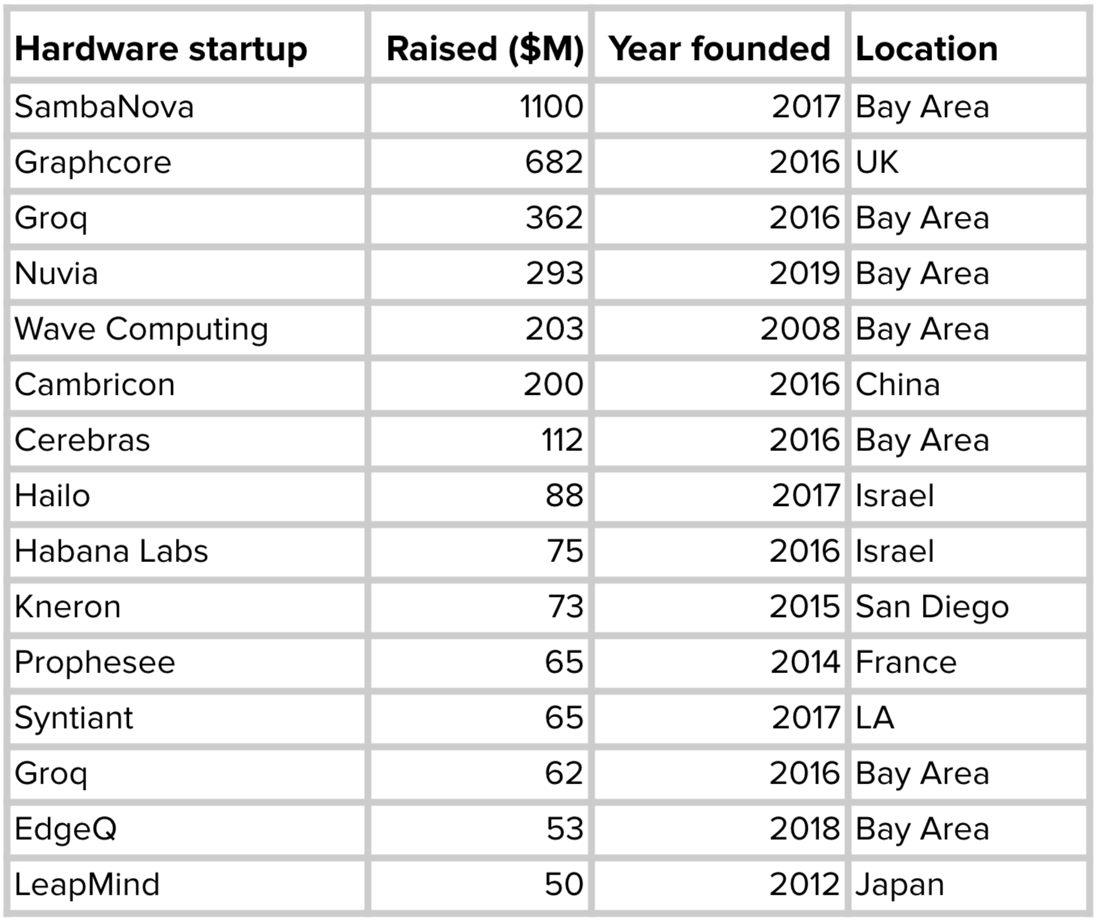
[22] Field Programmable Gate Arrays
[23] Application-Specific Integrated Circuits

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.

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, Tesla have all announced their plans to make their own chips. Meanwhile, ML hardware startups have raised billions of dollars to develop better AI chips (see Figure 6-11). It's projected that by 2025, the number of active edge devices worldwide will be over 30 billions.



Hardware startup	Raised ($M)	Year founded	Location
SambaNova	1100	2017	Bay Area
Graphcore	682	2016	UK
Groq	362	2016	Bay Area
Nuvia	293	2019	Bay Area
Wave Computing	203	2008	Bay Area
Cambricon	200	2016	China
Cerebras	112	2016	Bay Area
Hailo	88	2017	Israel
Habana Labs	75	2016	Israel
Kneron	73	2015	San Diego
Prophesee	65	2014	France
Syntiant	65	2017	LA
Groq	62	2016	Bay Area
EdgeQ	53	2018	Bay Area
LeapMind	50	2012	Japan

Figure 6-11: Notable hardware startups that have raised money as of September 2021. Fund-raising information by CrunchBase.

With so many new offerings for hardware to run ML models on, one question arises: how do we make your model run on arbitrary hardware efficiently? See Figure 6-12. 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.

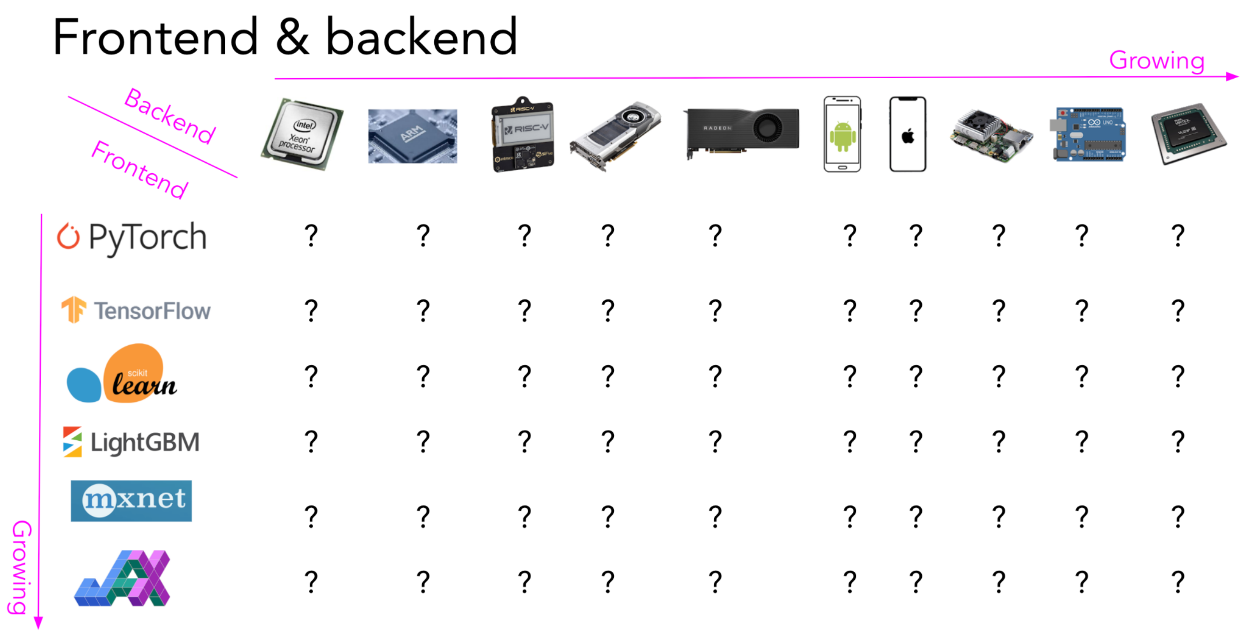


Figure 6-12: Given a growing number of frameworks and hardware backends, it's challenging to make an arbitrary framework on an arbitrary hardware backend.

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 6-13.

For example, the compute primitive of CPUs used to be a number (scalar), the compute primitive of GPUs used to be a one-dimensional vector, whereas the compute primitive of TPUs is a two-dimensional vector (tensor)[24]. Performing a convolution operator will be very different with 1-dimensional vectors compared to 2-dimensional vectors. Similarly, you'd need to take into account different L1, L2, and L3 layouts and buffer sizes to use them efficiently.

[24] Nowadays, many CPUs these days have vector instructions and some GPUs have tensor cores, which are 2-dimensional.

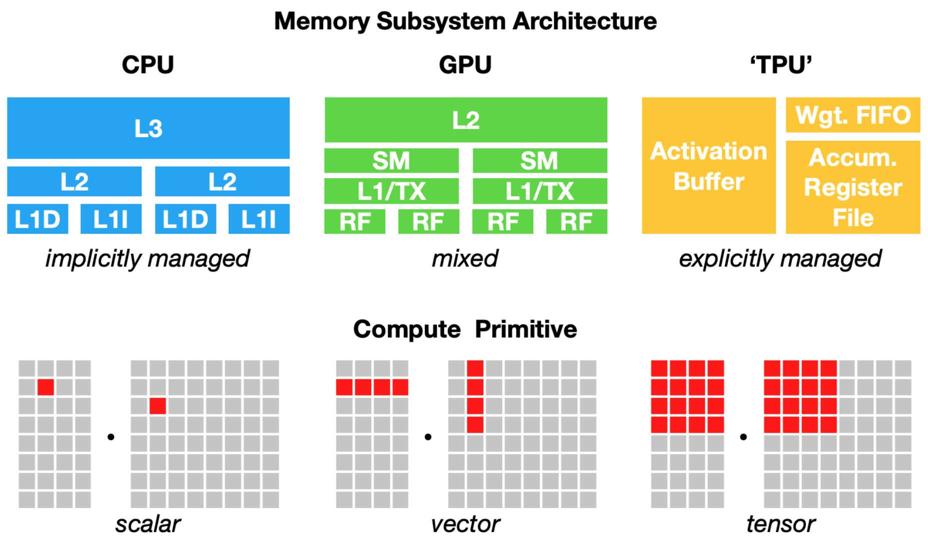


Figure 6-13: Different compute primitives and memory layouts for CPU, GPU, and TPU. Image by Chen et al., 2018.
[PERMISSION GRANTED]

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 middle man to bridge frameworks and platforms? Framework developers will no longer have to support every type of hardware, only need to translate their framework code into this middle man. Hardware vendors can then support one middle man instead of multiple frameworks.

This type of "middle man" 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 intermediate representations before generating the code native to a hardware backend so that it can run on that hardware backend, as shown in Figure 6-14.

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.

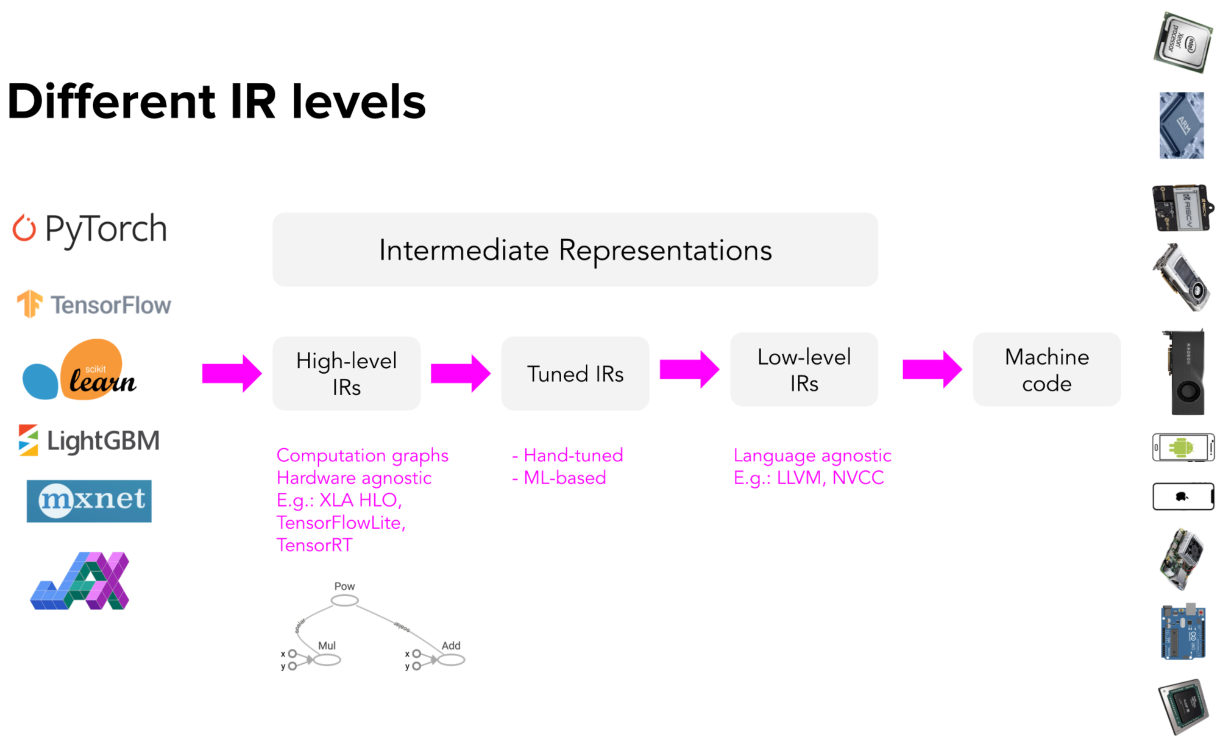


Figure 6-14: A series of high- and low-level IRs between the original model code to machine code that can run on a given hardware backend

High-level IRs are usually computation graphs of your ML models. For those familiar with TensorFlow, the computation graphs here are similar to the computation graphs you have encountered in TensorFlow 1.0, before TensorFlow switched to eager execution. In TensorFlow 1.0, TensorFlow first built the computation graph of your model before running it. This computation graph allows TensorFlow to understand your model's structure to optimize its runtime.

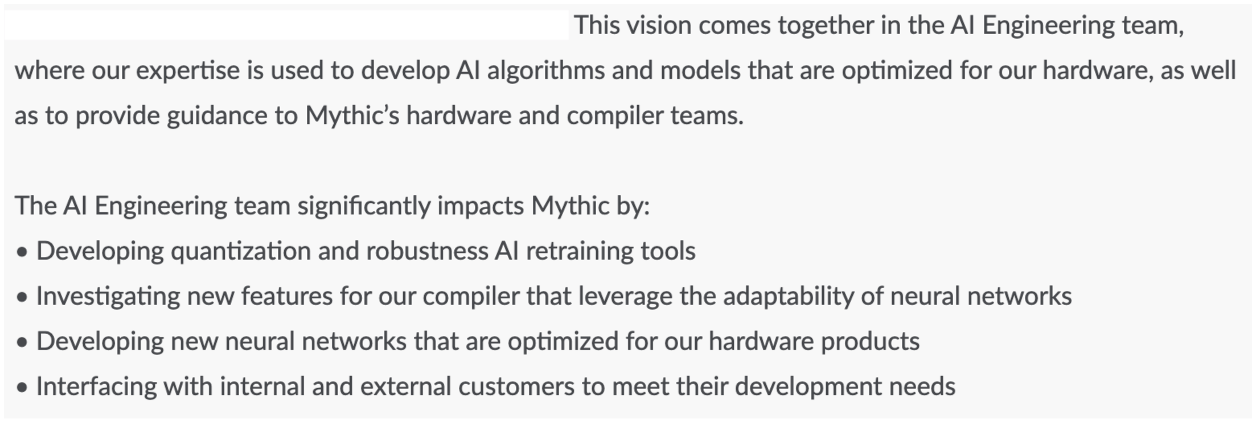
Model Optimization

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 tree model like LightGBM to generate features, then make predictions using an ensemble of models built with various frameworks like sklearn, TensorFlow, or HuggingFace's transformers.

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 (Palkar et al., '18).

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 is shown in Figure 6-15.


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

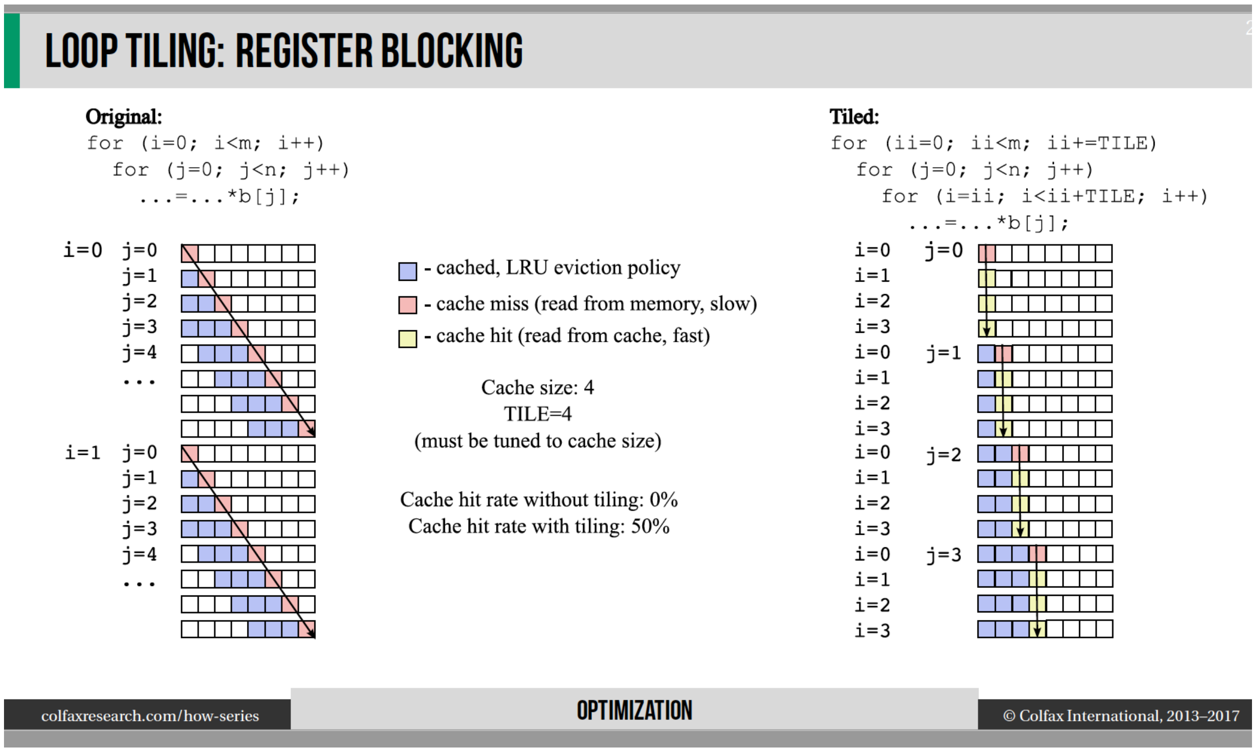
Figure 6-15: A job description for optimization engineers at Mythic.

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) is 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, and instead of executing it one item at a time, use hardware primitives to operate on multiple elements contiguous in memory.
- **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**: 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. A visualization of loop tiling is shown in Figure 6-16.
- **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 a single loop. An example of operator fusion is shown in Figure 6-17.



LOOP TILING: REGISTER BLOCKING

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```

Tiled:

```
for (ii=0; ii<m; ii+=TILE)
  for (j=0; j<n; j++)
    for (i=ii; i<ii+TILE; i++)
      ...=...*b[j];
```

🟦 - cached, LRU eviction policy

🟥 - cache miss (read from memory, slow)

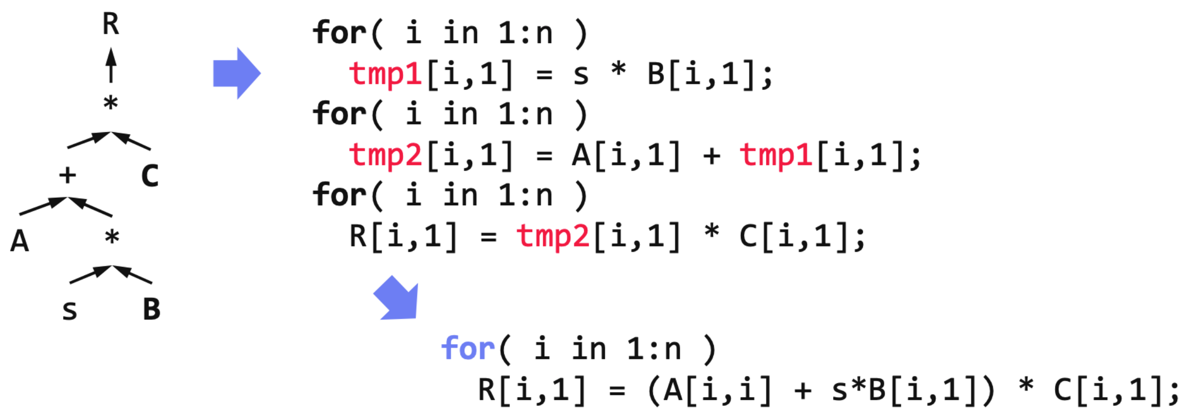
🟨 - cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

colfaxresearch.com/how-series OPTIMIZATION © Colfax International, 2013–2017

Figure 6-16: A visualization of loop tiling. Image by Colfax Research.
[TODO: WAITING PERMISSION]



```
for( i in 1:n )
  tmp1[i,1] = s * B[i,1];
for( i in 1:n )
  tmp2[i,1] = A[i,1] + tmp1[i,1];
for( i in 1:n )
  R[i,1] = tmp2[i,1] * C[i,1];
```

```
for( i in 1:n )
  R[i,1] = (A[i,i] + s*B[i,1]) * C[i,1];
```

Figure 6-17: An example of an operator fusion. Example by Matthias Boehm.
[PERMISSION GRANTED]

To obtain a much bigger speedup, you'd need to leverage higher-level structures of your computation graph. For example, given 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 6-18.

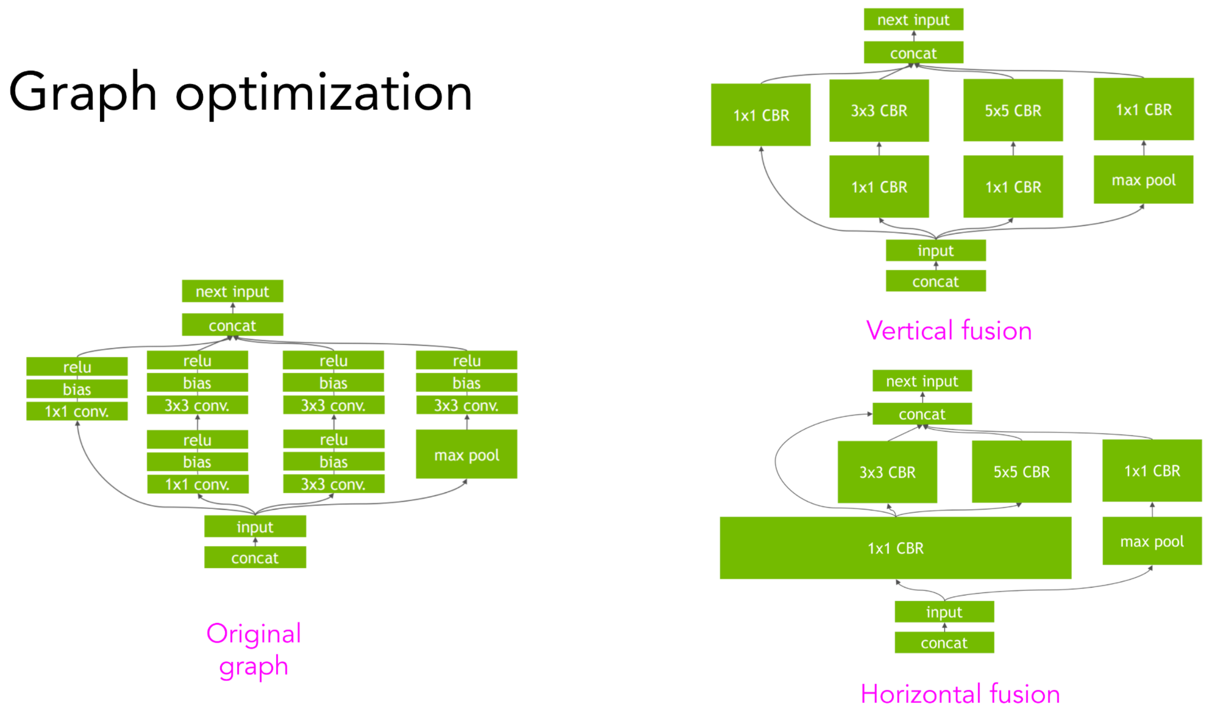


Figure 6-18: Vertical and horizontal fusion of the computation graph of a convolution neural network. Illustration by TensorRT.
[TODO: Need to reproduce]

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 3 operators A, B, and C, you can fuse A with B, fuse B with C, or fuse A, B, and C altogether.

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 focuses exclusively on how to make ResNet-50 run really fast on their DGX A100 server.[25]

There are a couple of drawbacks to hand-designed heuristics. First, they're non-optimal. There's no guarantee that the heuristics an engineer comes up with are the best possible solution. Second, they are non-adaptive. 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 ResNet50 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.

[25] 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.

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 6-19 shows the performance gain that autoTVM gave compared to cuDNN for the model ResNet-50 on NVIDIA TITAN X.

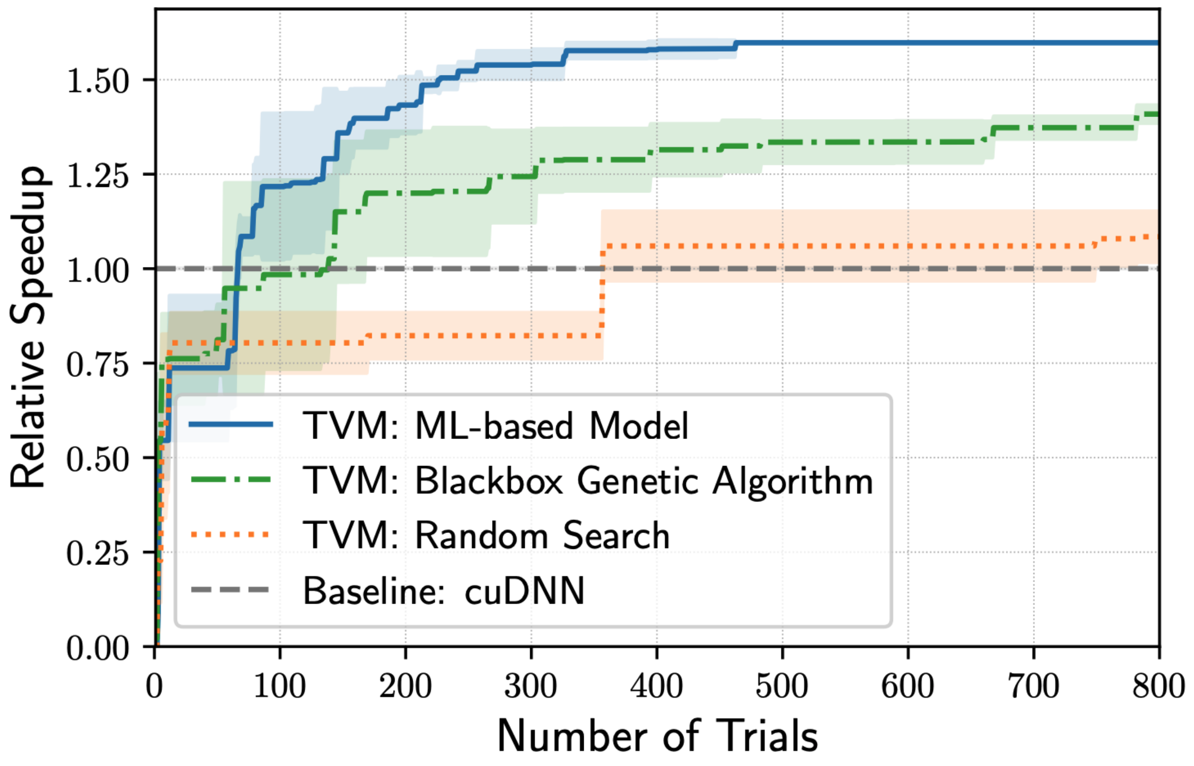


Figure 6-19: Speed up achieved by autoTVM over cuDNN for ResNet-50 on NVIDIA TITAN X. It takes ~70 trials for autoTVM to outperform cuDNN.

Experiment by Chen et al.
[PERMISSION GRANTED]

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 backend. This sort of optimization is ideal when you have a model ready for production, and target hardware to run inference on.

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 much 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 sklearn, 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 [1, 2]. As of September 2021, it's supported by 93% of devices worldwide.

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).

Summary

Congratulations, you've finished possibly the most technical chapter 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 6-20.

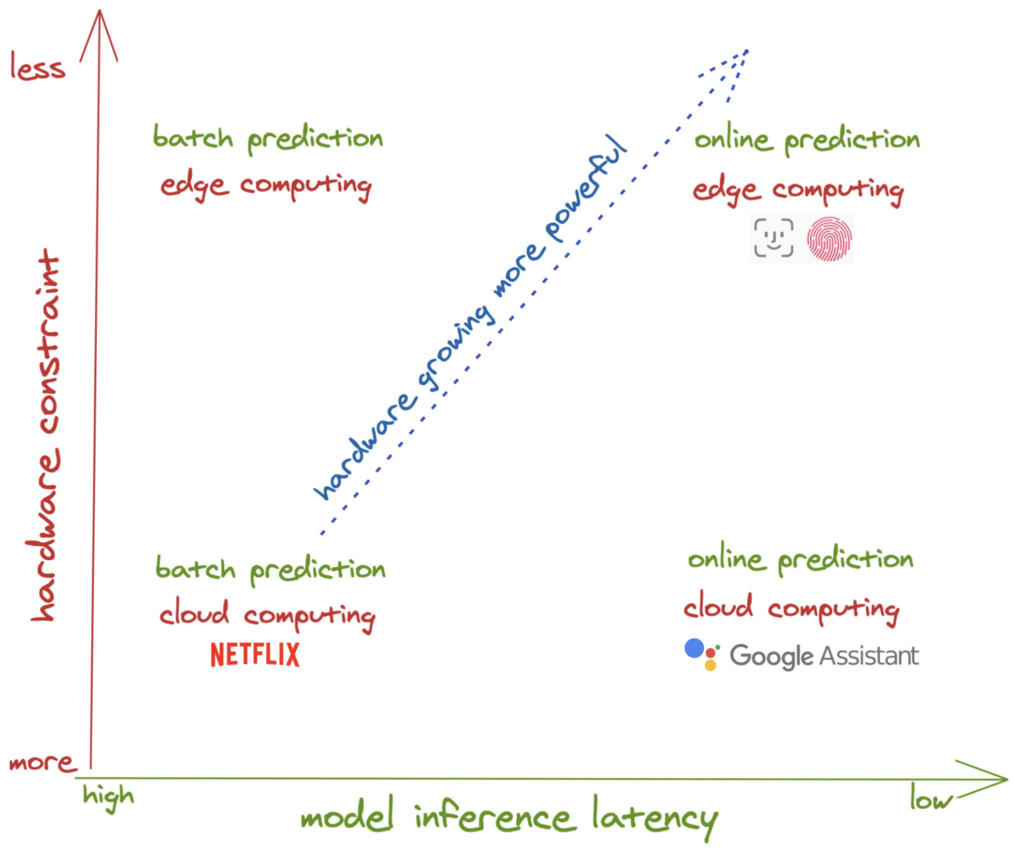


Figure 6-20: As hardware becomes more powerful, ML models will move to online and on the edge.

In addition to making predictions on-device, people are also working on techniques that enable ML model training over edge devices, such as the efforts around federated learning. And in addition to online predictions, people are also working to allow models to continually learn in production. We'll cover continual learning and federated learning in the next chapter.