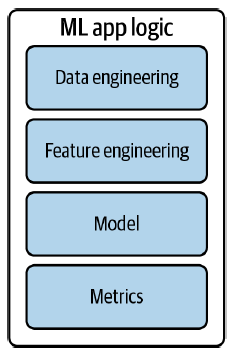
# Designing Machine Learning Systems - Chip Huyen

## Chapter 7 – Model Deployment and Prediction Service

* One of the most technical chapters in this book because **deploying ML models is an engineering challenge, not an ML challenge**
* 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)



* Developing this logic **requires both ML knowledge and SME**
* In many companies, this is the **part of the process that is done by the ML or data science teams**
* Another **part in the iterative process =** **deploying your model**
* **“Deploy” = a loose term that generally means making your model running and accessible**
* **During model development**, your **model usually runs in a DEV environment**
* To be **deployed**, your model will have to ***leave* DEV** + it can be **deployed to a staging environment for testing or to a production environment to be used by your end users**
* **Production is a spectrum**
* Some teams = production means generating nice plots in notebooks to show to business teams
* Other teams = production means keeping models up and running for millions of users a day
* “Deploying is easy if you ignore all the hard parts”
* If you want to deploy a model for 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**, and push your model + 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

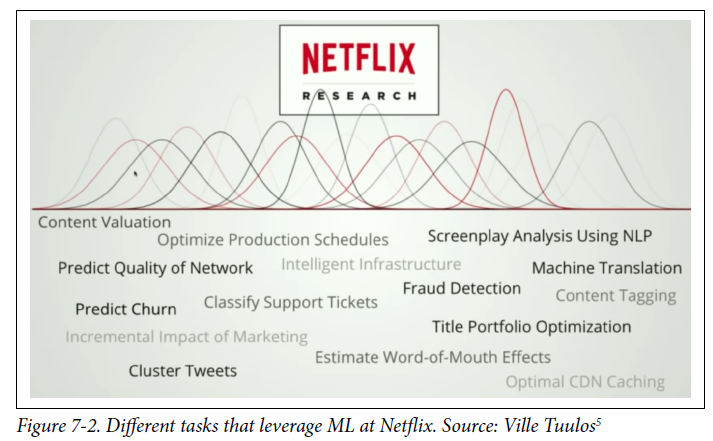
* **Can use this exposed endpoint for downstream applications** (e.g., when an application receives a prediction request from a user, this request is sent to the exposed endpoint, which returns a prediction
* If familiar with the necessary tools, you can have a functional deployment in an hour
* The **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**
* **Exporting** a model means **converting this model into a format that can be used by another application** (Some call this process **serialization**)
* There are **2 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** (ex: 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 + help you tailor them to their purposes**
* There are some common myths about ML deployment, often from people who haven’t deployed ML models
* There are **2 main ways a model generates + serves its predictions to users: online prediction and batch prediction**
* The **process of generating predictions** is called **inference**.
* **Where should the computation for generating predictions be done? On the device (also referred to as the edge) or the cloud?**
* **How a model serves + computes the predictions influences how it should be designed, the infrastructure it requires, and the behaviors that users encounter**

### ML Deployment Myths

* 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

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

* Academic projects = advised to choose a small problem to focus on, which usually leads to a single model
* Thinking of ML production in the context of a single model means **the infrastructure you 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, + each might require its own model**
* Ex: Uber needs a model to predict each of the following elements:
* Ride demand, driver availability, estimated time of arrival, dynamic pricing, fraudulent transaction, customer churn, + more
* Additionally, if this app operates in 20 countries, until you can have models that generalize across different user-profiles, cultures, + 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
* See the wide range of the tasks that leverage ML at Netflix:



* In fact, Uber has thousands of models in production, at any given moment, Google has thousands of models training concurrently with hundreds of billions parameters in size, and booking.com has 150+ models
* A 2021 Algorithmia study shows that **among organizations with > 25,000 employees, 41% have more than 100 models in production**

#### 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** **data distribution shifts** = **when the data distribution your model encounters in production is different from the data distribution it was trained on**
* 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

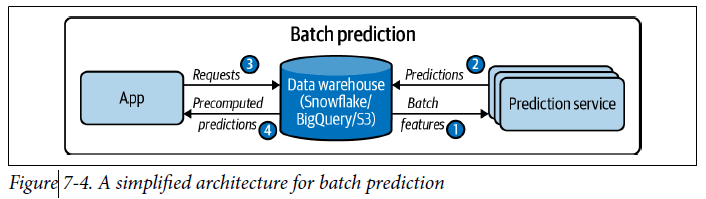
* The right question to ask is not “How often *should* I update my models?”, but instead **“How often *can* I update my models?”**
* **Since a model’s performance decays over time, we want to update it as fast as possible**
* This is an area of ML where we should **learn from existing DevOps best practices.**
* Even back in 2015, people were already constantly pushing out updates to their systems.
* Etsy deployed 50 times/day, Netflix 1000’s of times per day, AWS every 11.7 seconds
* 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 (similar numbers come from companies like Alibaba and ByteDance (TikTok))

#### Myth 4: Most MLE’s Don’t Need to Worry About Scale

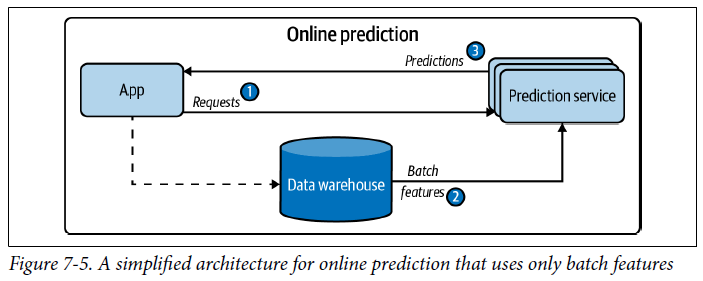
* **What “scale” means varies from application to application**, but examples include a system that serves hundreds of queries per second or millions of users a month.
* You might argue that, if so, only a small number of companies need to worry about it, there is only one Google, one Facebook, one Amazon
* True, but **a small number of large companies employ the majority of the SWE workforce**
* Stack Overflow Developer Survey 2019: > 1/2 of respondents work for a company of >= 100 employees
* So, a company of 100 employees has a good chance of serving a reasonable number of users
* **If 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**

### Batch Prediction vs. Online Prediction

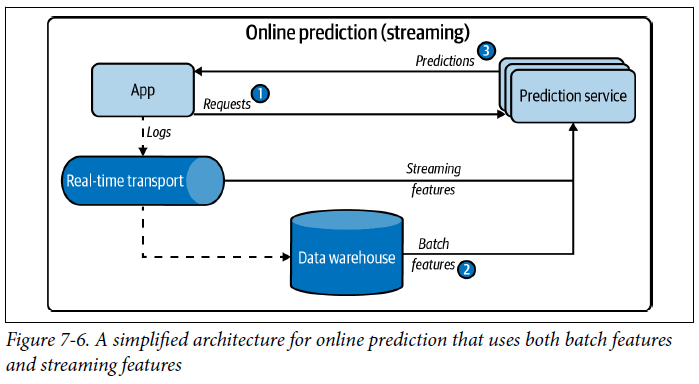
* **One fundamental decision you’ll have to make that will affect both your end users *and* developers working on your system = how it generates and serves its predictions to end users: online or batch**
* The terminologies surrounding **batch and online prediction are still quite confusing due to the lack of standardized practices in the industry**
* 3 main modes of prediction:
* **Batch prediction**, which **uses only batch features**.
* **Online prediction** that **uses only batch features** (e.g., **precomputed embeddings**).
* **Online/streaming prediction** that uses ***both* batch features and streaming features**
* **Online prediction**= **when predictions are generated + returned as soon as requests for these predictions arrive**
* Ex: Enter an English sentence into Google Translate, get back its French translation immediately
* 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)**
* 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** = **predictions are generated periodically or whenever triggered**
* The **predictions are stored somewhere** (SQL tables or an in-memory database, etc.) and **retrieved *as needed***
* Ex: Netflix might generate movie recommendations for all of its users every 4 hours, and the precomputed recommendations are fetched and shown to users when they log on
* 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 1 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**
* Here is a simplified architecture for batch prediction



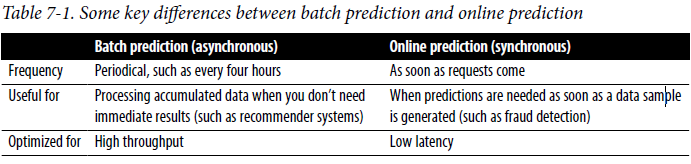
* Here is a simplified version of online prediction using only batch features



* **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**
* Ex: For example, after a user puts in an order on DoorDash, they might need the following features to estimate the delivery time:
* Batch features = 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
* Can hear terms “streaming features” and “online features” used interchangeably, but they are actually different
* **Online features = 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 + batch features:



* **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** = you **precompute predictions for popular** **queries**, then **generate predictions online for less popular queries**
* The table below summarizes the key points to consider for online prediction and batch prediction:



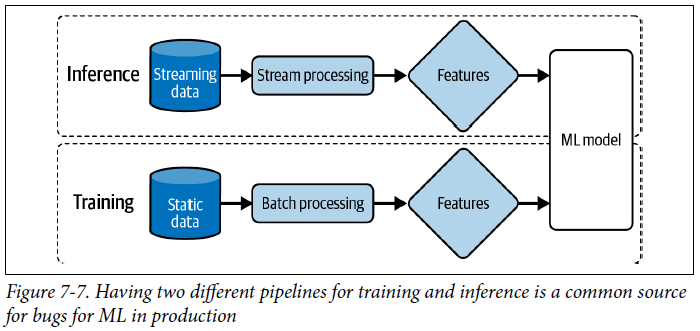
* In **many applications, online prediction and batch prediction are used side by side for different use cases**
* Ex: Food ordering apps (DoorDash, UberEats, etc.) use batch prediction to generate restaurant recommendations (it’d take too long to generate these recommendations online because there are many restaurants)
* However, once you click on a restaurant, food item recommendations are generated using online prediction
* **Many believe 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, for 2 reasons**
* **1) Streaming technologies like Apache Flink are proven to be highly scalable and fully distributed, which means they can do computation in parallel**
* **2) The strength of stream processing is in stateful computation.**
* Also, with online prediction, you don’t have to generate predictions for users who aren’t visiting your site
* Ex: 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 predictions will be wasted

#### From Batch Prediction to Online Prediction

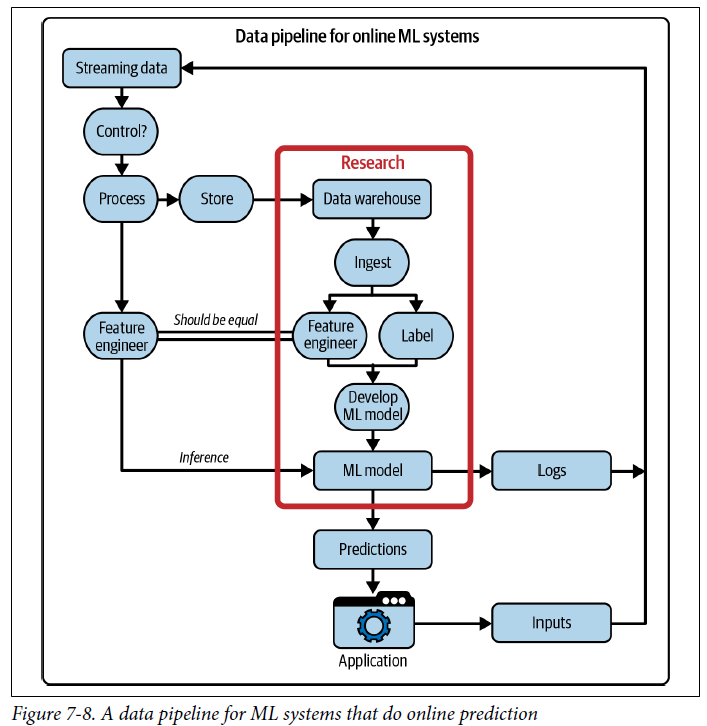
* To people coming to ML from an academic background, the more natural way to serve predictions is probably **online = give a model an input + it generates a prediction as soon as it receives that input**
* This is likely **how most people interact with their models while prototyping**
* This is also **likely easier to do for most companies when first deploying a model** = export a model, upload exported model to Amazon SageMaker or Google App Engine, and get back an **exposed endpoint** (**URL of the entry point for a service, which, in this case, is the prediction service of your ML model**)
* 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, and you don’t have to use all the predictions generated**
* Ex: 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 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?
* As of writing 2022, 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 (If a new user joins, you can give them some generic recommendations.)
* 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**
* Ex: For Netflix, batch prediction causes mild inconvenience (which is tightly coupled with user engagement and retention), not catastrophic failures
* But, 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 a phone using face/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 the fraudulent transaction from going through
* **Batch prediction = a workaround for when online prediction isn’t cheap or 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, 2 components are required**:
* **1) 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
* **2) A model that can generate predictions at a speed acceptable to its end users**
* For most consumer apps, this means milliseconds

#### 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**
* Ex: 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 5 minutes, and 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 2 different pipelines to process your data is a common cause for bugs in ML production**.
* One cause for **bugs = when the changes in one pipeline aren’t correctly replicated in the other, leading to 2 pipelines extracting 2 different sets of features**
* Especially common if the 2 pipelines are maintained by 2 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 below



* Below 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 academia



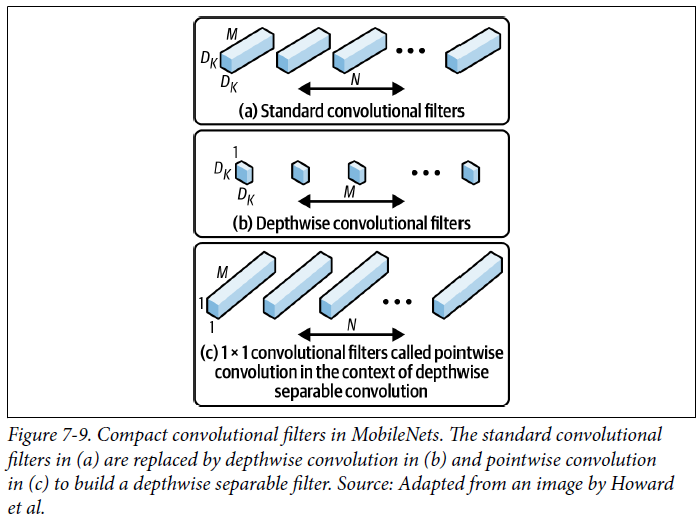
* **Building infrastructure to unify stream processing and batch processing has become a popular topic in recent years for the ML community**
* Companies including Uber and Weibo have made major infrastructure overhauls to unify their batch and stream processing pipelines by using a **stream processor** like Apache Flink
* Shuyi Chean and Fabian Hueske, “Streaming SQL to Unify Batch & Stream Processing w/ Apache Flink @Uber,”
* Yu, “Machine Learning with Flink in Weibo”
* **Some companies use feature stores to ensure the consistency between the batch features used during training and the streaming features used in prediction**

### 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**
* We **want fast inference for ML models**
* If the model you want to deploy takes too long to generate predictions, there are **3 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**
* The number of research papers on model compression is growing, and off-the-shelf utilities are proliferating
* While many new techniques being developed, the **4 types of techniques you might come across the most often** = **low-rank optimization, knowledge distillation, pruning, and quantization**

#### 1) Low-Rank Factorization

* The key idea behind **low-rank factorization****= to replace high-dimensional tensors with lower-dimensional tensors**
* One type of low-rank factorization is **compact convolutional filters = over-parameterized (having too many parameters) convolution filters are replaced with compact blocks to both reduce the number of parameters and increase speed**
* Ex: 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
* Similarly, MobileNets decomposes the standard convolution of size K×K×Cinto a depth-wise convolution (K×K×1) and a pointwise convolution (1×1×C), with Kbeing the kernel size and Cbeing the number of channels
* This means that each new convolution uses only K2 + Cinstead of K2Cparameters
* If K = 3, this means an 8-9X times reduction in the number of parameters (see below)



* **Low-rank factorization 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 CNNs) and **requires a lot of architectural knowledge to design, so it’s not widely applicable to many use cases yet**

#### 2) Knowledge Distillation

* **Knowledge distillation =a small model (student) is trained to mimic a larger model or ensemble of models (teacher), and *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**
* Ex: One distilled network used in production is DistilBERT = reduces the size of a BERT model by 40% while retaining 97% of its language understanding capabilities and being 60% faste
* The **advantage of this approach = it can work regardless of the architectural differences between the teacher and the student networks**
* Ex: Can get a random forest as the student and a transformer as the teacher
* The **disadvantage of this approach = 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,** which **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**

#### 3) Pruning

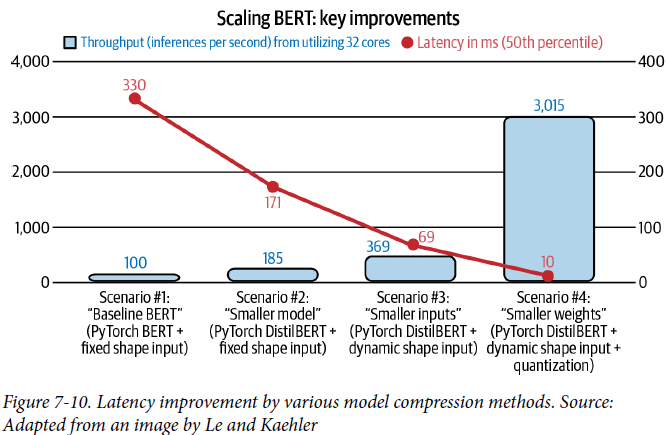
* **Pruning**= a method originally used for decision trees where you remove sections of a tree that are uncritical and redundant for classification
* As NN’s gained wider adoption, people started to realize that NN’s are over-parameterized and began to find ways to reduce the workload caused by the extra parameters
* **Pruning, in the context of neural networks, has 2 meanings**
* **1) To remove entire nodes**, which means changing the NN’s architecture and reducing its number of parameters
* **2) To find parameters least useful to predictions and set them to 0** (a more common meaning)
* In this case, pruning doesn’t reduce the total number of parameters, *only the number of nonzero parameters*, and **the architecture of the NN remains the same**
* This **helps with reducing the size of a model because pruning makes a NN 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**
* BUT **pruning can introduce biases into your model.**
* *While it’s generally agreed that pruning “works”, 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 dense counterpart

#### 4) Quantization

* **Quantization****reduces a model’s size by using fewer bits to represent its parameters**
* The **most general and commonly used model compression method**
* It’s **straightforward to do and generalizes over tasks and architectures**. Quantization
* 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, which take 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 NN’s) like BinaryConnect and XNOR-Net
* **Quantization not only reduces memory footprint but also improves the computation speed**
* **1) It allows us to increase our batch size**
* **2) Less precision speeds up computation = further reduces training time and inference latency**
* Consider the addition of 2 numbers
* If we perform the addition bit by bit, and each takes x nanoseconds, it’ll take 32x nanoseconds for 32-bit numbers but only 16xnanoseconds for 16-bit numbers
* There are **downsides to quantization**:
* **Reducing the number of bits to represent numbers means 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**
* Also run the risk of rounding/scaling your numbers to under-/overflow 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**
* **Quantization can either happen during training (quantization aware training), 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 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
* 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.”
* **Training in fixed-point is not yet as popular but has had a lot of promising results**
* **Fixed-point inference has become a standard in the industry**, and **some edge devices *only* support fixed-point inference**
* Most popular frameworks for on-device ML inference (Google TensorFlow Lite, Facebook PyTorch Mobile, NVIDIA TensorRT) offer post-training quantization for free in 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*
* For many of their NLP services, they needed to handle over 25,000 inferences per second at a latency of under 20 milliseconds, as shown below
* 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 i



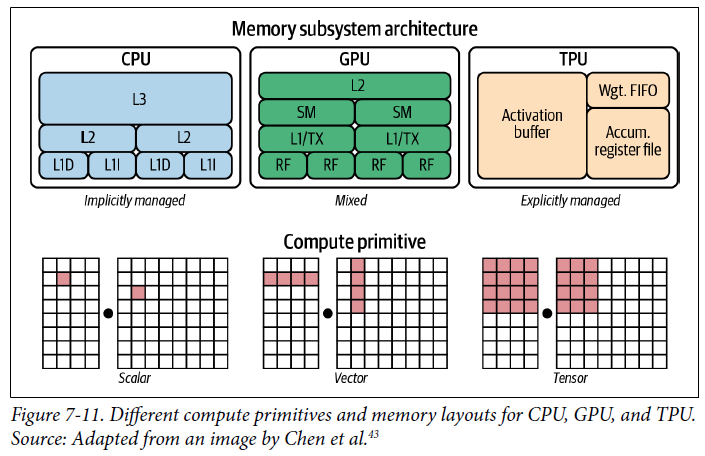
* **The biggest performance boost they got came from quantization**
* Converting 32-bit floating points to 8-bit integers reduces latency 7X + increases throughput 8X
* **Results here seem very promising to improve latency; however, they should be taken with a grain of salt as there’s no mention of changes in output quality after each performance improvement**

### ML on the Cloud and on the Edge

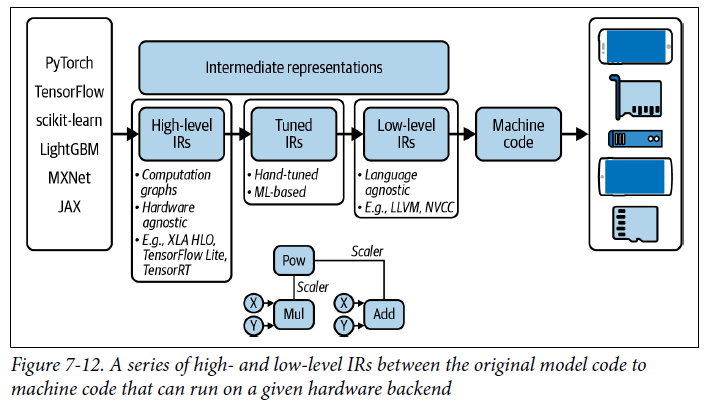
* Another decision you’ll want to consider is **where your model’s computation will happen: on the cloud or on the edge**
* **On the cloud = a large chunk of computation is done on the cloud, either public or private clouds**
* **On the edge = a large chunk of computation is done on consumer (edge) devices** (browsers, phones, laptops, smartwatches, cars, security cameras, robots, embedded devices, FPGAs (field programmable gate arrays), and ASICs (application-specific integrated circuits)
* The **easiest way is to package your model up and deploy it via a managed cloud service such as AWS or GCP (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**
* **1) Cost, since 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
* That number for small and medium companies can be between $50K and $2M a year
* A mistake in handling cloud services can cause startups to go bankrupt
* **As 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**
* **1) 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 models to work in situations with no internet connections or where connections are unreliable, such as in rural areas or developing countries
* **2) When your models are already on consumers’ devices, you can worry less about network latency**
* Requiring data transfer over the network (sending data to a model on the cloud to make predictions then sending predictions back to users) might make some use cases impossible
* **In many cases, network latency is a bigger bottleneck than inference latency**
* Ex: Might be able to reduce the inference latency of ResNet-50 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
* **3) 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**
* Edge computing makes it easier to comply with regulations, like GDPR, about how user data can be transferred or stored
* BUT 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 even 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 plans to make their own chips
* Meanwhile, ML hardware startups have raised billions of dollars to develop better AI chips
* Projected that by 2025 the number of active edge devices worldwide will be > 30 billion
* 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?**

#### 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**
* Ex: 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 + 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 below



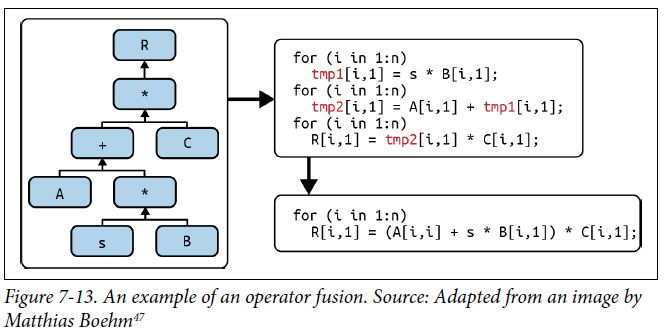
* Ex: The compute primitive of CPUs used to be a number (scalar), the compute primitive of GPUs used to be a 1D vector, whereas the compute primitive of TPUs is a 2D vector (tensor) (*Nowadays, many CPUs have vector instructions and some GPUs have tensor cores, which are 2D*)
* Performing a convolution operator will be very different with 1D vectors compared to 2D 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)**, which **lies 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 below



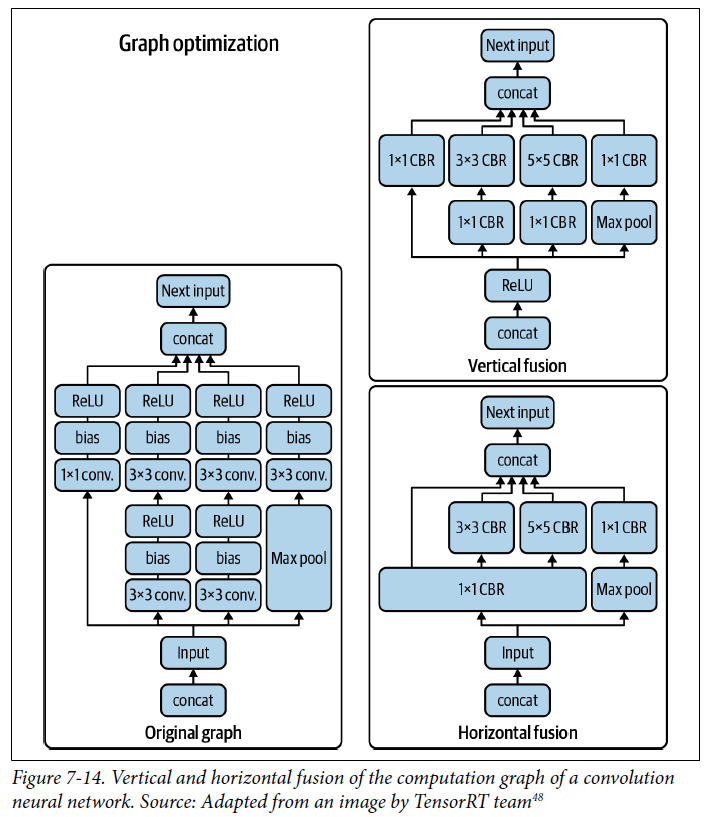
* This process is also called **lowering**, **as in you “lower” your high-level framework code into low-level hardware-native code**
* It’s ***NOT* translating because there’s no one-to-one mapping between them**
* **High-level IRs are usually computation graphs of your ML models** = **a graph that describes 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***
* It **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**
* *Might use pandas/dask/ray to extract features from data, NumPy to perform vectorization, a pretrained model like Hugging Face’s Transformers to generate features, then make predictions using an ensemble of models built with various frameworks (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 *23X slower* in one thread compared to *hand-optimized code*
* **In many companies, what usually happens is that data scientists and MLE’s 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**
* Ex: 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 NN’s
* 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 + the operators it consists of (convolution, loops, cross-entropy, etc.) and find a way to speed it up
* There are **2 ways to optimize your ML models: locally and globally**
* **Locally = you optimize an operator or a set of operators of your model**
* **Globally = 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**
* 4 of the local optimization common techniques:
* **1) 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
* **2) Parallelization**
* Given an input array (or n-dimensional array), divide it into different, independent work chunks, and do the operation on each chunk individually
* **3) 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*
* **4) Operator fusion**
* Fuse multiple operators into one to avoid redundant memory access
* Ex: 2 operations on the same array require 2 loops over that array
* In a fused case, it’s just one loop
* Below is an example of operator fusion.

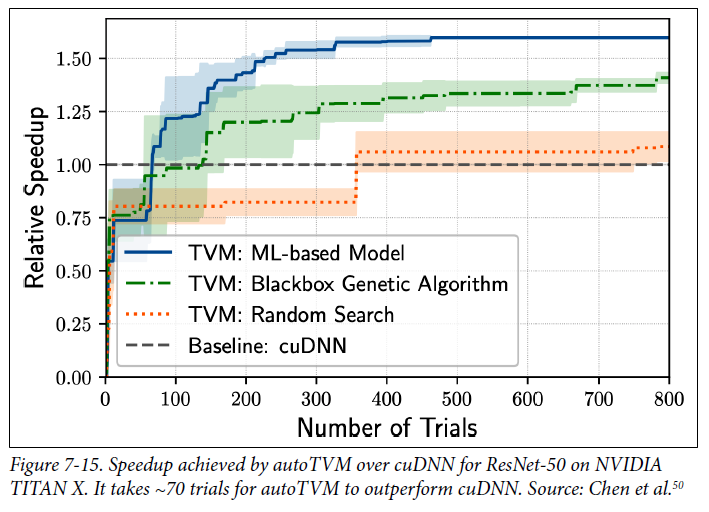


* **To obtain a much bigger speedup, you’d need to leverage higher-level structures of your computation graph**
* Ex: A CNN with the computation graph can be fused vertically *or* horizontally to reduce memory access and speed up the model, as shown below



##### Using ML to optimize ML models

* Like with the vertical and horizontal fusion for a CNN**, there are many possible ways to execute a given computation graph**
* Ex: Given 3 operators A, B, C 🡪 you can fuse A with B, fuse B with C, or fuse A, B, + 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**
* Ex: NVIDIA might have an engineer/team of engineers who focus exclusively on how to make ResNet-50 run really fast on their DGX A100 server
* *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 same hardware 🡪 It might just be that this model is over-optimized**
* There are a couple of **drawbacks to hand-designed heuristics**
* **1) They’re nonoptimal**
* There’s **no guarantee** that heuristics an engineer comes up with are the **best possible solution**
* **2) 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 CNN is different from optimizing a RNN, 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, which searches over a predetermined set of options to execute a convolution operator + 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**
* Below shows the performance gain autoTVM gave compared to cuDNN for the model ResNet-50 on NVIDIA TITAN



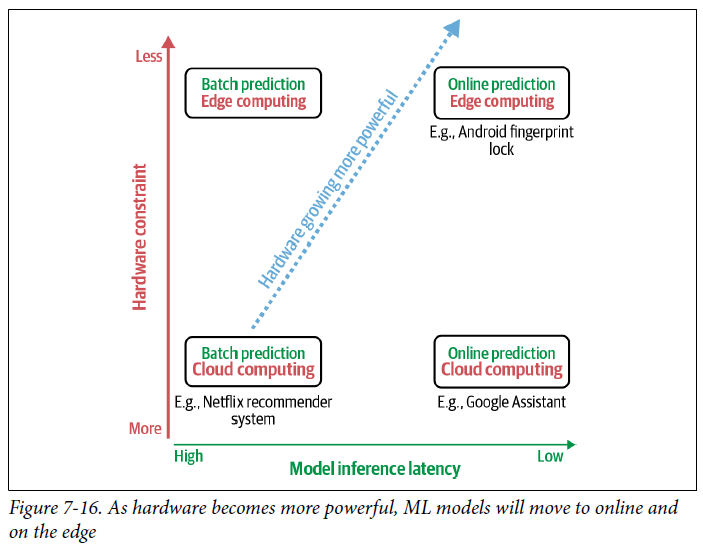
* **While the results of ML-powered compilers are impressive, they have a catch: they can be slow**
* You **go through all the possible paths and find the most optimized ones**
* This process **can take hours, even days for complex ML models**
* *However*, it’s **a one-time operation, and results of 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**

#### 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 compile models into JavaScript 🡪 TensorFlow.js, Synaptic, 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) = an open standard that allows you to run executable programs in browsers**
* After you’ve built your models in *whatever* frameworks you’ve used (scikit-learn, PyTorch, TensorFlow, etc.), **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 in the last couple of years
* It’s **performant, easy to use, and has an ecosystem that is growing like wildfire**
* *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

* **Deploying ML models is an *engineering* challenge, not an ML challenge**
* Different ways to deploy a model = comparing online prediction with batch prediction, and ML on the edge with ML on the cloud, each way having 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**
* **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, ML systems should be able to transition to making online prediction on-device**, illustrated below



* May used to think that an ML project is done after the model is deployed, but this is a serious mistake
* **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**
* **Models might fail in production**, and we have to **continually monitor models to detect issues and address them as fast as possible**