Codementor Blog Last updated May 08, 2019

Machine Learning: How to Build Scalable Machine Learning Models

Now that you understand why scalability is needed for machine learning and what the benefits are, we'll do a deep dive into the various solutions that address the frequent problems and bottlenecks we may face while developing a scalable machine learning pipeline.

This post will cover:

- Picking the right framework/language
- Using the right processors
- Data collection and warehousing
- The input pipeline
- Model training
- Distributed machine learning
- Other optimizations
- Resource utilization and monitoring
- Deploying and real-world machine learning

As before, you should already be familiar with concepts like neural network (NN), Convolutional Neural Network (CNN), and ImageNet.

Picking the Right Framework and Language

There are many options available when it comes to choosing your machine learning framework. While your gut feeling might be to just go with the best framework available in the language of your proficiency, this might not always be the best idea.

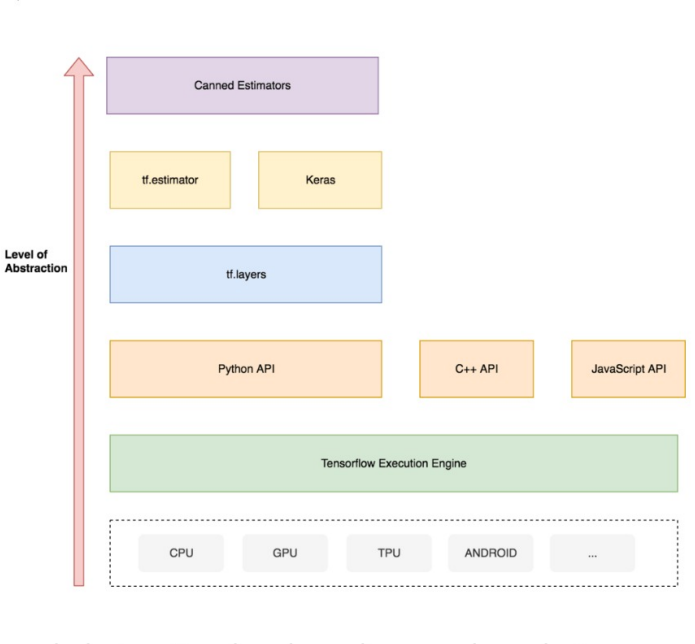
For example, the use of Java as the primary language to construct your machine learning model is highly debated. Standard Java lacks hardware acceleration. One may argue that Java is faster than other popular languages like Python used for writing machine learning models. The thing to note is that most machine learning libraries with Python interface are wrappers over C/C++ code, which make them faster than native Java.

To achieve comparable performance with Java, we will need to wrap some C/C++/Fortran code. There are implementations which do that, but very few as compared to other languages. Moreover, since machine learning involves a lot of experimentation, the absence of REPL and strong static typing, make Java not so suitable for constructing models in it.

Beyond language is the task of choosing a framework for your machine learning solution. Some of the popular deep learning frameworks are TensorFlow, Pytorch, MXNet, Caffe, and Keras.

There are multiple factors to consider while choosing the framework like community support, performance, third-party integrations, use-case, and so on. We won't go into what framework is best; you can find a lot of nice features and performance comparisons about them on the internet. However, one important thing to keep in mind while selecting the library/framework is the **level of abstraction you want to deal with**.

For example, consider this abstraction hierarchy diagram for TensorFlow:



Your preferred abstraction level can lie anywhere between writing C code with CUDA extensions to using a highly abstracted canned estimator, which lets you do a lot (optimize, train, evaluate) with fewer lines of code but at the cost of less control on implementation. It mostly depends on the complexity and novelty of the solution that you intend to develop.

Next up: Using the right processors | Data collection and warehousing | The input pipeline | Model training | Distributed machine learning | Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

Using the Right Processors

CPUs, GPUs, ASICs, and TPUs

Since a large part of machine learning is feeding data to an algorithm that performs heavy computations iteratively, the choice of hardware also plays a significant role in scalability. Scaling activities for computations in machine learning (specifically deep learning) should be concerned about executing matrix multiplications as fast as possible with less power consumption (because of cost!).

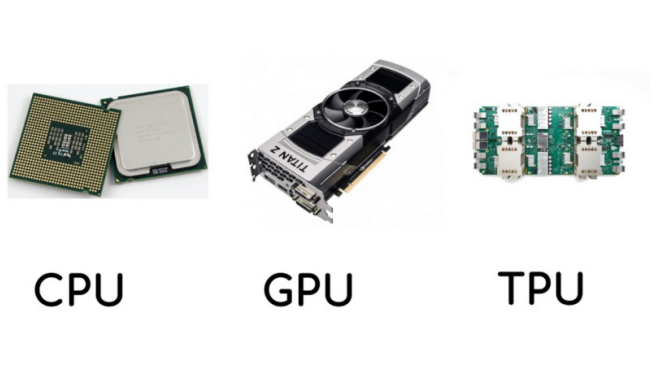
CPUs are not ideal for large scale machine learning (ML), and they can quickly turn into a bottleneck because of the sequential processing nature. An upgrade on CPUs for ML is GPUs (graphics processing units). Unlike CPUs, GPUs contain hundreds of embedded ALUs, which make them a very good choice for any process that can benefit by leveraging parallelized computations. GPUs are much faster than CPUs for computations like vector multiplications. However, both CPUs and GPUs are designed for general purpose usage and suffer from von Neumann bottleneck and higher power consumption.

A step beyond CPUs and GPUs is ASICs (Application Specific Integrated Chips), where we trade general flexibility for an increase in performance. There have been a lot of exciting research on for designing ASICs for deep learning, and Google has already come up with three generations of ASICs called Tensor Processing Units (TPUs).

TPUs exploit the fact that neural network computations are operations of matrix multiplication and addition, and have the specialized architecture to perform just that. TPUs consist of MAC units (multipliers and accumulators) arranged in a systolic array fashion, which enables matrix multiplications without memory access, thus consuming less power and reducing costs.

This way of performing matrix multiplications also reduces the computational complexity from the order of `n3` to order of `3n - 2`. Explaining how they work is beyond the scope of this article, but you can read more about that here.

Google is not the only player in this domain though, other tech companies like Huawei, Microsoft, Facebook, and Apple have also been actively working on designing ASICs for machine learning.



To sum it up, CPUs are scalar processors, GPUs are vector processors, and ASICs like TPUs are matrix processors.

Next up: Data collection and warehousing | The input pipeline | Model training | Distributed machine learning | Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

Data Collection and Warehousing

Data collection and warehousing can sometimes turn out to be the step with the most human involvement. Activities like cleaning, feature selection, labeling can often be redundant and time-consuming. To reduce the effort in labeling and also to expand data, there has been active research going on in the area of producing synthetic data using generative models like GANs, Variational Autoencoders, and Autoregressive models. The downside is that these models require very high computation to be able to generate synthetic data, and it's not as helpful as real-world data.

The format in which we're going to store the data is also vital. It mostly depends on the kind of data that we're dealing with, and how we're going to use it. For instance, if you have to feed the data to a distributed architecture, then formats like HDF5 can be quite efficient.

Next up: The input pipeline | Model training | Distributed machine learning | Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

The Input Pipeline

I/O hardware are also important for machine learning at scale. The massive data on which we iteratively perform computations is fetched from and stored by I/O devices. With hardware accelerators, the input pipeline can quickly become a bottleneck if not optimized. It can broadly be seen as consisting of three steps:

1. **Extraction**: The first task is to read the source. The source can be a disk, a stream of data, a network of peers, etc.

2. **Transformation**: We might need to apply some transformations to the data. For example, in the case of training an image classifier, transformations like resizing, flip, cross, rotate, and grayscale are applied to the input image before feeding them to the model.

3. **Loading**: The final step bridges between the working memory of the training model and the transformed data. Those two locations can be the same or different depending on what kind of devices we are using for training and transformation.

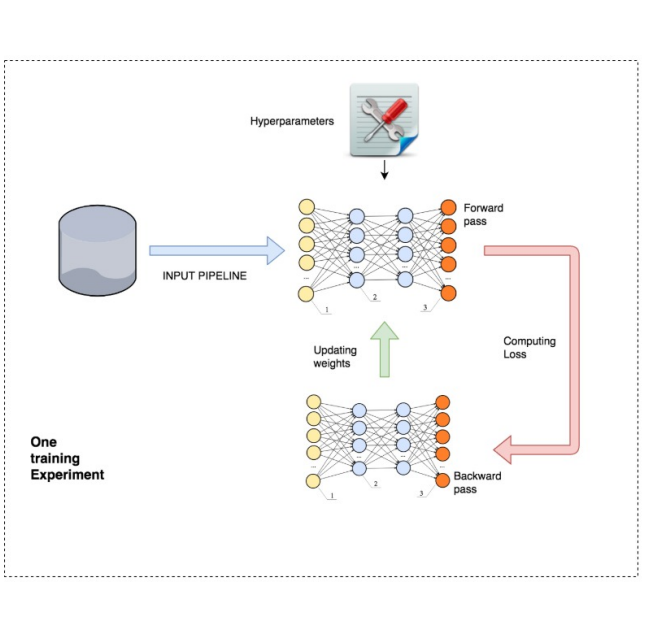
Now we can see that all three steps rely on different computer resources. Extraction depends on I/O devices (reading from disk, network, etc.); transformation usually depends on CPU; and assuming that we are using accelerated hardware, loading depends on GPU/ASICs.

We can take advantage of this fact, and break down input data into batches and parallelize file reading, data transformation, and data feeding. This way we can interweave the three steps and optimize resource utilization, so that none of the steps are blocked due to dependency on the other.

Next up: Model training | Distributed machine learning | Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

Model Training

Coming to the core step of a Machine learning pipeline, if we would like to see training step at a slightly more detailed level, here's how it'll look like.



A typical, supervised learning experiment consists of feeding the data via the input pipeline, doing a forward pass, computing loss, and then correcting the parameters with an objective to minimize the loss. Performances of various hyperparameters and architectures are evaluated before selecting the best one.

Let's explore how we can apply the "divide and conquer" approach to decompose the computations performed in these steps into granular ones that can be run independently of each other, and aggregated later on to get the desired result. After decomposition, we can leverage horizontal scaling of our systems to improve time, cost, and performance.

Next up: Distributed machine learning | Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

Dimensions of decomposition

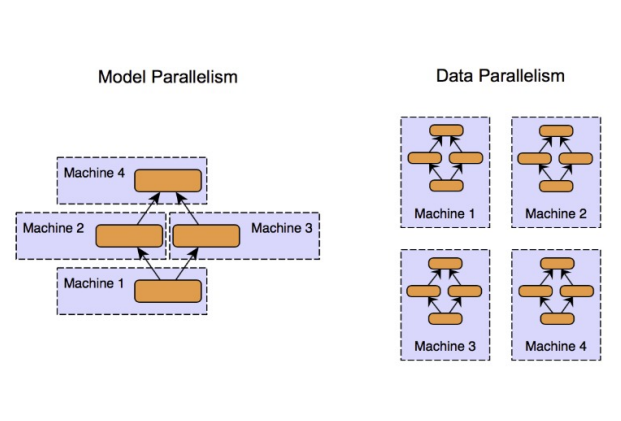
There are two dimensions to decomposition: functional decomposition and data decomposition.

Functional decomposition

Functional decomposition generally implies breaking the logic down to distinct and independent functional units, which can later be recomposed to get the results. "Model parallelism" is one kind of functional decomposition in the context of machine learning. The idea is to split different parts of the model computations to different devices so that they can execute in parallel and speed up the training.

Data decomposition

Data decomposition is a more obvious form of decomposition. Data is divided into chunks, and multiple machines perform the same computations on different data.



One instance where you can see both the functional and data decomposition in action is the training of an ensemble learning model like random forest, which is conceptually a collection of decision trees. Decomposing the model into individual decision trees in functional decomposition, and then further training the individual tree in parallel is known as data parallelism. It is also an example of what's called embarrassingly parallel tasks.

Based on the idea of functional and data decomposition, let's now explore the world of distributed machine learning. You know, all the big data, Spark, and Hadoop stuff that everyone keeps talking about? We can leverage that for machine learning as well!

Distributed Machine Learning

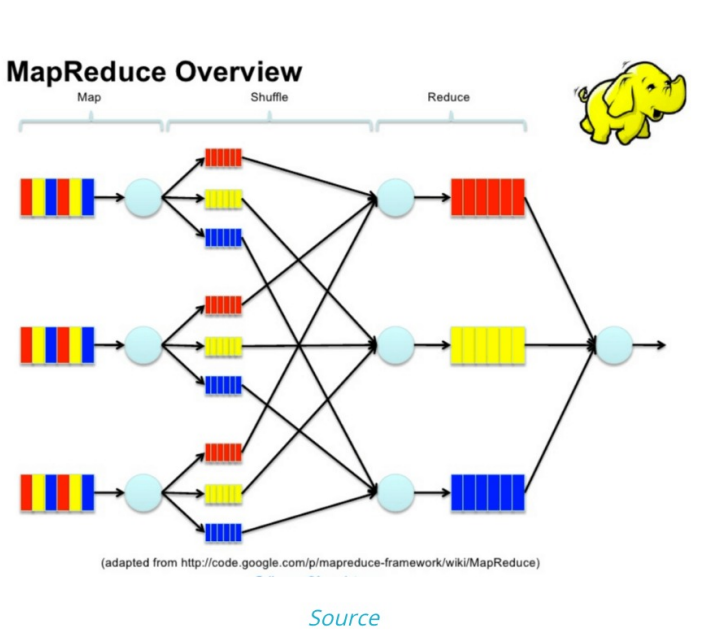
Decomposition in the context of scaling will make sense if we have set up an infrastructure that can take advantage of it by operating with a decent degree of parallelization. And when we talk about this, an important question to seek an answer to is "How do we express a program that can be run on a distributed infrastructure?"

Next up: Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

MapReduce paradigm

MapReduce is a programming model built to allow parallelization of computations. The model is based on "split-apply-combine" strategy. A typical MapReduce program will express a parallelizable process in a series of map and reduce operations. The map function maps the data to zero or more key-value pairs. The MapReduce execution framework groups these key-value pairs using a shuffle operation. Then, the reduce function takes in those key-value groups and aggregates them to get the final result.

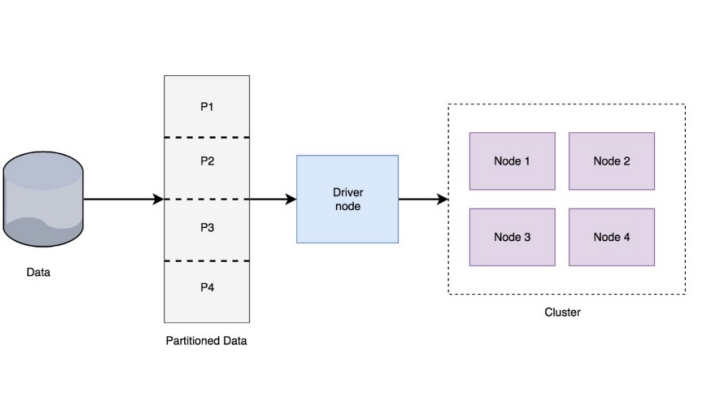
The thing to note is that the MapReduce execution framework handles data in a distributed manner, and takes care of running the Map and Reduce functions in a highly optimized and parallelized manner on multiple workers (aka cluster of nodes), thereby helping with scalability.


MapReduce Overview
Map
Shuffle
Reduce
(adapted from http://code.google.com/p/mapreduce-framework/wiki/MapReduce)

Source

Distributed machine learning architecture

Let's talk about the components of a distributed machine learning setup.



The data is partitioned, and the driver node assigns tasks to the nodes in the cluster. The nodes might have to communicate among each other to propagate information, like the gradients. There are various arrangements possible for the nodes, and a couple of extreme ones include Async parameter server and Sync AllReduce.

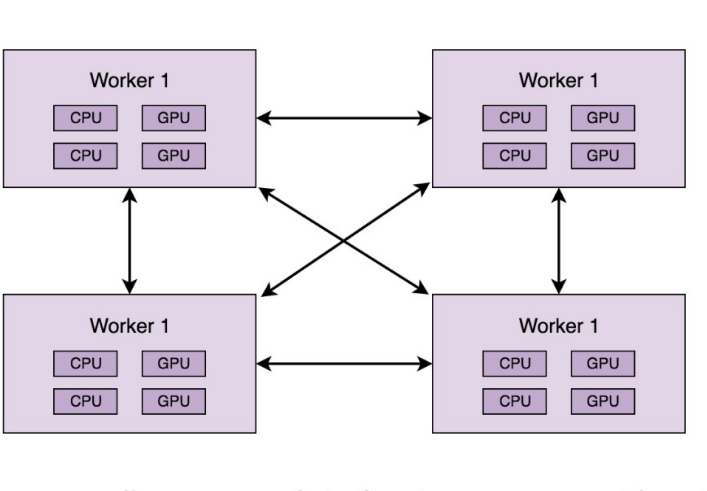
Async parameter server architecture

In the Async parameter server architecture, as the name suggests, the transmission of information in between the nodes happens asynchronously. Here's a typical architecture diagram for this type of architecture:

You can see how a single worker can have multiple computing devices. The worker, labeled "master", also takes up the role of the driver. Here the workers communicate the information (e.g. gradients) to the parameter servers, update the parameters (or weights), and pull the latest parameters (or weights) from the parameter server itself. One drawback of this kind of set up is delayed convergence, as the workers can go out of sync.

Sync AllReduce architecture

In a Sync AllReduce architecture, the focus is on the synchronous transmission of information between the cluster node. Here's a typical architecture diagram for Sync AllReduce architecture:



Workers are mutually connected via fast interconnects. There's no parameter server. This kind of arrangement is more suited for fast hardware accelerators. All workers have to be synced before a new iteration, and the communication links need to be fast for it to be effective.

Some distributed machine learning frameworks do provide high-level APIs for defining these arrangement strategies with little effort.

Popular frameworks for distributed machine learning

A distributed computation framework should take care of data handling, task distribution, and providing desirable features like fault tolerance, recovery, etc. The most popular open-source implementation of MapReduce is **Apache Hadoop**. Hadoop stores the data in the Hadoop Distributed File System (HDFS) format and provides a Map Reduce API in multiple languages. The scheduler used by Hadoop is called YARN (Yet Another Resource Negotiator), which takes care of optimizing the scheduling of the tasks to the workers based on factors like localization of data.

Another popular framework is **Apache Spark**. Spark's design is focused on performing faster in-memory computations on streaming and iterative workloads. Spark uses immutable Resilient Distributed Datasets (RDDs) as the core data structure to represent the data and perform in-memory computations. For machine learning with Spark, we can write our algorithms in the MapReduce paradigm, or we can use a library like MLlib.

Spark is very versatile in the sense that you can run Spark using its standalone cluster mode, on EC2, Hadoop YARN, Mesos, or Kubernetes. And there's a support for accessing data in HDFS, Apache Cassandra, Apache HBase, Apache Hive, and lots of other data sources.

Another Apache framework to consider is **Apache Mahout**. Mahout is more focused on performing distributed linear-algebra computations. Mahout also supports the Spark engine, which means it can run inline with existing Spark applications. It is easier to write or extend an algorithm in Mahout if it doesn't have an implementation in any Spark library like MLlib.

That being said, MapReduce isn't the only way to have your Machine learning algorithm executed in a distributed manner. Message Passing Interface (MPI) is another programming paradigm for parallel computing. MPI is a more general model and provides a standard for communication between the processes by message-passing. It gives more flexibility (and control) over inter-node communication in the cluster. For use cases involving smaller datasets or more communication amongst the decomposed tasks, libraries based on MPI can be a better choice.

All the mature deep learning frameworks like TensorFlow, MxNet, and PyTorch also provide APIs to perform distributed computations by model and data parallelism. Also, there are frameworks at higher-level like horovod and elephas built on top of these frameworks. So it all boils down to what your use-case is and what level of abstraction is appropriate for you.

Next up: Other optimizations | Resource utilization and monitoring | Deploying and real-world machine learning

Hyperparameter Optimization

When solving a unique problem with machine learning using a novel architecture, a lot of experimentation is involved with hyperparameters. The hyperparameter search space can be large, and it may not be practically feasible to try every combination. This can make the fine tuning process really difficult. The solution to this problem lies in using a hyperparameter optimization strategy to select the best (or approximately best) hyperparameters for the model.

Hyperparameter optimizations aim to minimize the loss function on a given set of data. One such technique that you are already familiar with is gradient-based optimization, which is used in training neural networks to find the ideal weights.

For other kinds of machine learning models like SVM, Decision trees, Q-learning, etc., we can try out other strategies like random search, Bayesian optimization, and evolutionary optimization. A couple of popular frameworks for hyperparameter optimization in a distributed environment are Ray and Hyperopt.

Other optimizations

Memory efficient backpropagation

The memory requirements for training a neural network increases linearly with depth and the batch size. There have been active research to diminish this linear scaling so that memory usage can be reduced. arXiv:1604.06174 proposes a technique for square root scaling instead of linear at the cost of a little extra computation complexity. The Openai/gradient-checkpointing package implements an extended version of this technique so that you can use it in your TensorFlow models.

Low precision training

This is another area with a lot of active research. Machine learning frameworks, by default, use 32-bit floating point precision for inference and training the models. There is evidence that we can use to lower numerical precision (like 16-bit for training, and 8-bit for inference) at the cost of minimal accuracy.

Reducing the precision will right away lead to reduced memory footprint, better bandwidth utilization, improved caching, and sped up model performance (hardware can perform more operations per second on low precision operands). However, reducing precision is not as straightforward as simply casting all the values to lower precision. It leads to quantization noise, gradient underflow, imprecise weight updates, and other similar problems.

If you want to dig deeper on how to do it correctly, Nvidia's documentation about mixed precision training is highly recommended.

Other things worth considering

It's easy to get lost in the sea of emerging techniques for efficiently doing machine learning at scale. We should also keep the following things in mind while judiciously designing our architecture and pipeline:

- In most use cases, **accuracy is not the only thing that matters**. After certain iterations of training, the extra-accuracy that we gain on every new training iteration tends to become negligible. We shouldn't strive too hard for those minimal accuracy gains and know when it's the right time to pull the brakes and stop training.
- Unless we are working on a problem statement that hasn't been solved before, or trying to come up with a novel architecture, we should **try *not* to reinvent the wheel**. For instance, for a natural language processing task like sentiment analysis, we can download already existing vector embeddings like GloVe, and for image classification tasks we can use pre-trained state-of-the-art models like VGG-16 trained on Imagenet. We can always fine-tune these pre-trained models for our use case using transfer learning.
- Try to **look for distributed versions of algorithms**. Elastic SGD, Asynchronous SGD, Butterfly SGD, and Sparse SGD — there isn't a one-technique-fits-all for solving scaling problems in machine learning. Not everything can be parallelized; some algorithms do not have a distributed implementation. So it's vital to spend extra effort during the conceptual modeling phase to plan the architecture wisely.
- When doing machine learning models at scale, it becomes vital to **track versions and history of the models** (e.g. which model was used, what hyperparameters were used, how previous iterations performed, and so on). This helps to know, in the long run, which model was chosen over the others and why.

Next up: Resource utilization and monitoring | Deploying and real-world machine learning

Resource utilization and monitoring

When you're training at scale, it's important to actively monitor different aspects of the pipeline for memory and CPU usage. Using cloud services like elastic compute be a double-edged sword (in terms of cost) if not used carefully. It's always advisable to run a mini version of your pipeline on a resource that you completely own (like your local machine) before starting full-fledged training on the cloud.

Also, to get the most out of available resources, we can interweave processes depending on different resources so that no resource is idle (e.g. interweaving extraction, transformation, and loading in the input pipeline). Many companies have also designed internal orchestration frameworks responsible for the scheduling of different machine learning experiments in an optimized way.

Deploying and Real-world Machine Learning

Here comes the final part, putting the model out for use in the real world. The first thing to consider is **how to serialize your model**. Most frameworks have high-level APIs for checkpointing (or saving) and loading models. And if you do end up using some custom serialization method, it's a good practice to separate the architecture (algorithm) and the coefficients (parameters) learned during training.

There might also be various kinds of use cases for the trained model. You might have to integrate it inside an existing software, or maybe you want to expose it to the web.

If the idea is to expose it to the web, then there are a few interesting options to explore. For instance, you can execute a TensorFlow/Keras model on the user's browser with TensorFlow.js, which is a WebGL based library for deploying/training ML models that also supports hardware acceleration. This way you won't even need a back-end. The downsides is that your model is publically visible (including the weights), which might be undesirable in some cases, and the inference time depends on the client's machine.

If you are planning to have a back-end with an API, then it all boils down to how to scale a web application. We can consider using a typical web server architecture with a load balancer (or a queue mechanism) and multiple worker machines (or consumers).

We can also consider a serverless architecture on the cloud (like AWS lambda) for running the inference function, which will hide all the operationalization complexity and allow you to pay-per-execution. One caveat with AWS Lambda is the cold start time of a few seconds, which by the way also depends on the language.

Since cold start happens **once** for every concurrent execution request, if your application traffic is spikey in nature and can strictly tolerate much less latency, then it might not be the best option. On the other hand, if traffic is predictable and delays in very few responses are acceptable, then it's an option worth considering.

Finally, there are other full-fledged services like Amazon SageMaker, Google Cloud ML, and Azure ML that you might want to have a look at. Apart from the usual cloud web features like auto-scaling, you'll get machine learning specific features like the auto-tuning of hyperparameters, monitoring dashboards, easy deployments with rolling updates, and well-defined pipelines. However, the downside is the ecosystem lock-in (less flexibility) and a higher cost.

Conclusion

In this two post series, we analyzed the problem of building scalable machine learning solutions. We went through a lot of technologies, concepts, and ongoing research topics relevant to doing machine learning at scale. We tried to cover a lot of breadths and just-enough depth. We hope that the next time you face the challenge of implementing a machine learning solution at scale, you'll know what to do!

Published May 07, 2019