Lecture 24: Machine Learning Accelerators.

CS4787 — Principles of Large-Scale Machine Learning Systems

So far, we've talked about machine learning running on two types of classical hardware: CPUs and GPUs. But these are not the only options for training and inferring ML models. An exciting new generation of computer processors is being developed to accelerate machine learning calculations. These so-called **machine learning accelerators** (also called AI accelerators) have the potential to greatly increase the efficiency of ML tasks (usually deep neural network tasks), for both training and inference. Beyond this, even the traditional-style CPU/GPU architectures are being modified to better support ML and AI applications. Today, we'll talk about some of these trends.

Overview of hardware in ML. As we saw in the first lecture, the ML pipeline has many different components.

- Data collection, cleaning, labeling, and maintenance
- Training
- Hyperparameter optimization
- Testing and validation
- Inference and deployment
- ...*et cetera*

Hardware can help improve performance pretty much everywhere in the pipeline, and there's interest from hardware vendors in designing better hardware for pretty much every aspect of the ML pipeline. Two main ways to do this:

- Adapt existing architectures to ML tasks.
- Develop brand-new architectures for ML.

Both of these methods have seen significant use in recent years.

What improvements can we hope to get from better hardware in the ML pipeline?

What does this mean for the statistical performance of our algorithms?

One major issue when developing new hardware for ML, and the question we should always be asking: **how is the device programmed?**

- To be useful for DNN applications, we should be able to map a high-level program written in something like TensorFlow/PyTorch/MxNet/Keras to run on the hardware.
- Ideal scenario: users don't need to reason about the hardware that their code is running on. They just see the speedup.
- Current state-of-the-art: users still need to give some thought to the hardware.
  - For example, if the accelerator uses low-precision arithmetic, the performance of an algorithm in full precision on a CPU may differ from the performance of the algorithm in low-precision on the accelerator.
  - But it is possible to get TensorFlow code to run on an accelerator with minimal modifications!

GPU as ML accelerator. One important thing to realize is that the only real distinction between GPUs and ML accelerators is that GPUs weren't originally designed for AI/ML. But there's nothing in the architecture itself that particularly separates GPUs from more purpose-built ML accelerators. In fact, as machine learning tasks capture more of the market for GPUs, GPU designers have been adjusting their architectures to fit ML applications.

- For example, by supporting low-precision arithmetic.
- For example, by creating specialized compute paths for tasks common to DNN architectures.
- For example, by making it easier for multiple GPUs to communicate with each other so as to collaborate together on a single training task.

As GPU architectures become more specialized to AI tasks, it becomes more accurate to think of them as ML accelerators.

FPGA as ML accelerator. All computer processors are basically integrated circuit: electronic circuits etched on a single piece of silicon. Usually this circuit is fixed when the chip is designed. A *field-programmable gate array* or FPGA is a type of chip that allows the end-user to reconfigure the circuit it computes in the field (hence the name).

- Note that this doesn't actually involve physical changes to the circuit that's actually etched on the physical silicon of the FPGA: that's fixed. Rather, the FPGA constructs a logical circuit that is reconfigurable by connecting or disconnecting various parts of the circuit that is etched on its silicon.
- FPGAs were used historically for circuit simulation.

FPGAs consist of an array of programmable circuits that can each individually do a small amount of computation, as well as a programmable *interconnect* that connects these circuits together. The large number of programmable gates in the FPGA makes it a naturally highly parallel device.

Why would we want to use an FPGA instead of a GPU/GPU?

- An important property of FPGAs that distinguishes them from CPUs/GPUs: you can choose to **have data flow through the chip however you want!**
  - Unlike CPUs which are tied to their cache-heirarchy-based data model, or GPUs which perform best under a streaming model.
- FPGAs often use less power to accomplish the same work compared with other architectures.
  - But they are also typically slower.

When would we want to use an FPGA vs. building our own *application-specific integrated circuit* (ASIC) from scratch?

- Pro for FPGA: Much cheaper to program and FPGA than the design an ASIC.
- Pro for FPGA: A single FPGA costs much less than the first ASIC you synthesize.
- Pro for FPGA: FPGA designs can be adjusted on the fly.
- Pro for ASIC: The marginal cost of producing an additional ASIC is lower if you really want to synthesize millions or billions of them.
- Pro for ASIC: ASICs can typically achieve higher speed and lower power.

Who uses FPGAs in machine learning?

- Main one is Microsoft's Project Capatult/Project Brainwave[1]. For example, their website milestone from 2017 says "MSR and Bing launched hardware microservices, enabling one web-scale service to leverage multiple FPGA-accelerated applications distributed across a datacenter. Bing deployed the first FPGA-accelerated Deep Neural Network (DNN). MSR demonstrated that FPGAs can enable real-time AI, beating GPUs in ultra-low latency, even without batching inference requests."

The Tensor Processing Unit. Google's Tensor Processing Unit (TPU) made a splash in 2015 as one of the first specialized architectures for machine learning and AI applications. The original version focused on fast inference via high-throughput 8-bit arithmetic.

- Most of the chip is dedicated to accelerating 8-bit integer dense-matrix-dense-matrix multiplies
  - Note that even though the numbers it multiples are in 8-bit, it uses 32-bit accumulators to sum up the results.
  - This larger accumulator is common in ML architectures that use low precision.
- ...with a little bit of logic on the side to apply activation functions.

The second- and third-generation TPUs are designed to also support training and can calculate in floating point.

[1] `https://www.microsoft.com/en-us/research/project/project-catapult/`

How do we program a TPU?

- Google supports running TensorFlow code on TPUs. This makes it easy to train and infer deep neural networks.
- In fact, you can now run your own programs on the TPU on Google Cloud.[2]

Why use a TPU instead of a CPU/GPU?

- Pro for TPU: Google has some evidence that the TPU outperforms GPUs and other accelerators on benchmark tasks. From Google's blog:[3] "For example, it's possible to achieve a 19% speed-up with a TPU v3 Pod on a chip-to-chip basis versus the current best-in-class on-premise system when tested on ResNet-50"
- Pro for TPU: Seems to have better power and somewhat better scalability than other options. E.g. you can scale up to 256 v3 TPUs in a pod.
- Con for TPU: It ties you to Google and Google's Cloud Platform. You can't own a TPU.

Other ML accelerators.

Intel's **Nirvana Neural Network Processor** (NNP).[4]

- From their website: "The Intel Nervana NNP is a purpose built architecture for deep learning. The goal of this new architecture is to provide the needed flexibility to support all deep learning primitives while making core hardware components as efficient as possible."
- Built in collaboration with Facebook.

Apple's Neural Engine within the A11 Bionic system-on-a-chip for neural networks on iPhones.

...and many others!

[2] https://cloud.google.com/tpu/
[3] https://cloud.google.com/blog/products/ai-machine-learning/mlperf-benchmark-establishes-that-google-cloud-offers-the-most-a
[4] https://www.intel.ai/intel-nervana-neural-network-processors-nnp-redefine-ai-silicon/

Summary and open questions.

Scaling machine learning methods is increasingly important. In this course, we addressed the high-level question: **What principles underlie the methods that allow us to scale machine learning?** To answer this question, we used techniques from three broad areas: statistics, optimization, and systems. We articulated three broad principles, one in each area.

- **Statistics Principle: Make it easier to process a large dataset by processing a small random subsample instead.**
- **Optimization Principle: Write your learning task as an optimization problem, and solve it via fast general algorithms that update the model iteratively.**
- **Systems Principle: Use algorithms that fit your hardware, and use hardware that fits your algorithms.**

▷ Open problem: **reproducibility and debugging of machine learning systems**.

- Most of the algorithms we discussed in class are randomized, and random algorithms are hard to reproduce.
- Even when we don't use explicitly randomized methods, floating point imprecision can still make results difficult to reproduce exactly.
  - For hardware efficiency, the compiler loves to reorder floating point operations (this is sometimes called fast math mode) which can introduce slight differences in the output of an ML system.
  - As a result, even running the same learning algorithm on the same data on different ML frameworks *can* result in different learned models!
- Reproducibility is also made more challenging when hyperparameter optimization is used.
  - Unless you have the random seed, it's impossible to reproduce someone else's random search.
  - Hyperparameter optimization provides lots of opportunity for (possibly unintentional) cheating, where the test set is used improperly.
- ML models are difficult to debug because they often **learn around bugs**.

▷ Open problem: **more scalable distributed machine learning**.

- Distributed machine learning has this fundamental tradeoff with the batch size.
  - Larger batch size good for systems because there's more parallelism.
  - Smaller batch size good for statistics because we can make more "progress" per gradient sample. (For the same reason that SGD is generally better than gradient descent.)
- Communication among workers is expensive in distributed learning.
  - Need provably robust ways of compressing this communication to use fewer bits.

- The datacenters of the future will likely have many heterogeneous workers available.
  - How can we best distribute a learning workload across heterogeneous workers?
- When running many workers in parallel, the performance will start to be bound by **stragglers**, workers that take longer to work than their counterparts. How can we deal with this while still retaining performance guarantees?

▷ Open problem: **robustness to adversarial examples**.

- It's easy to construct examples that fool a deep neural network.
- How can we make our scalable ML methods provably robust to these type of attacks.