Lecture 24: Inference, Deployment, and Compression.

CS4787 — Principles of Large-Scale Machine Learning Systems

Most of this class has been focusing on ways of efficiently *training* machine learning models. But for many ML systems, training is only half the story. After a model is trained, we still need to actually use it to make predictions and/or decisions. This is usually called **inference**.

Although many of the methods we've discussed for scaling up systems can be used for inference as well, there are also some methods that are specific to inference. But before we get to the techniques, we first need to ask: **how do we measure the performance of a deployed ML inference system?**

Here are some important metrics that we use in practice to evaluate deployed ML systems.

- **Accuracy.** How accurate is the model on the queries coming in?
  - If both the test set and the new queries are coming from the same distribution, then we expect this accuracy to be close to the test accuracy.
  - Depending on the application, this can be a loss or statistical performance metric other than accuracy (e.g. mean squared error).
- **Latency.** How long do we need to wait between when we make a query and when we get a prediction from the system?
- **Throughput.** If we have a large batch of queries we want a prediction for, how many predictions can we get per second out of the system?
- **Energy/power.** How much energy is consumed to make one prediction? How much power is used by the deployed ML system in general?
- **Model size.** How many bytes are needed to store and/or transmit the learned model?
- **Memory use.** How much memory is used by the deployed ML system?
- **Cost.** How much money does my deployment cost?

Important point: **when deploying an ML system, there are trade-offs among ALL of these metrics!**

- Need to decide which one we value most when designing the system.

CPU vs GPU. For deep learning training, it is widely accepted that GPUs are the way to go. But for inference, the best device to use is by no means clear. Generally speaking:

- GPUs have higher throughput, especially when running at scales where the full parallel capabilities of the GPU can be utilized.
- Latency can be application-dependent. For smaller networks, CPUs can have the edge here. (But this is something you should measure if you really care about latency.)
- CPUs generally cost less.
- CPU-like architectures also are usually a better way to go when low power is needed, since it's easier to put a very-low-power CPU in a mobile or IoT device. But many mobile chips now are CPU/GPU hybrids so the lines are blurred here.

Techniques that try to reduce model size by possibly trading-off accuracy are called **model compression** techniques.

- Usually, these techniques have the side effect of improving the latency, throughput, energy/power, and memory use as well.

What methods that we already learned about could be used to make a model smaller?

Model compression for Deep Neural Networks. Goal is to find a smaller, easier-to-compute network with the same or similar accuracy. There are many techniques for doing this.

▷ One technique: **low-precision inference**. Just use low-precision arithmetic for inference. This is a relatively well-studied technique that can work very well with limited extra effort from users.

How low to go? Simple heuristic: just keep lowering the number of bits until the validation accuracy starts to decrease. Usually this method will get you below 16 bits.

An extreme type of low-precision inference is **binarized or ternerized neural networks**, which use only a single bit (for the binarized case) or two-ish bits (for the ternarized case) for each weight.

▷ Another technique: **pruning**. Idea is to remove weights that are close to zero and/or remove activations that are usually zero. This effectively creates a *smaller model*.

- A simple heursitic for pruning: set a threshold, then remove all weights from the model if they are smaller in magnitude than that threshold.

Pruning is often done together with **fine-tuning**. Fine-tuning isn't as model compression technique *per se*, but rather a technique that's used *by* model compression. Fine-tuning just refers to re-training the network starting from some given parameters by running a relatively small number of epochs with a relatively low learning rate. Many model compression methods work by repeatedly performing some compression step followed by a fine-tuning step.

- For example, for pruning, we can prune (removing some weights from the network) then fine-tune. Importantly, the fine-tuning step is only operating on the weights that *weren't* pruned. The pruned weights are gone and will remain 0 forever.

- We can do a similar thing for low-precision inference by fine-tuning the low-precision model.

▷ Another technique: **old-school compression**. This is a simple technique of just applying a traditional lossless compression technique, such as Huffman coding, to the learned weights of a network. Even something like gzip-ping the weights could produce some improvement here. This lowers the model size without impacting anything, but we do need to decompress it at the other end so it can come at the cost of some compute.

▷ Another technique: **knowledge distillation**. The idea is to take a large and/or complex model (called the teacher) and train a smaller and/or simpler network (called the student) to match its output. Importantly, the smaller network is trained not only to match the prediction made by the larger network (which would be redundant with the training set anyway) but also the distribution of the output neurons of the larger network.

- This is often used for distilling *ensemble models*, models involving multiple networks whose predictions are averaged together, into a single easier-to-compute model.

Surprisingly, knowledge distillation can sometimes even *improve* accuracy relative to the original network. We can even sometimes see this when the same model architecture is used for the teacher and student networks.

▷ Another technique: **efficient architectures**. Some neural network architectures are just designed to be efficient at inference time.

- Examples: MobileNet, ShuffleNet, CirCNN.

These networks are usually based on structured sparse linear layers.

- such as convolutions

To use these efficiently, we just train one of these efficient architectures in the first place for our application. (So this is not really a model compression technique, strictly speaking!) But care is needed to choose an architecture that is appropriate for any given task.

The last resort for speeding up DNN inference. If your DNN is really too slow, and none of the above methods help, you have one last resort that often works wonders. **Just don't use a neural network.** There are many other faster types of machine learning model that can often perform nearly as well or better than a neural net.

- If you can get away with using a linear model, that's almost always going to be faster for inference.

- Decision trees are also great for inference speed.

Where do we run inference? When we deploy our ML models, where do they run?

Most common place: **in the cloud**. The cloud supports large amounts of compute, makes it easy to scale with demand, and makes it easy and reliable to access compute. This is a good place to put AI that needs to interact with large amounts of data. It's also a good place to put compute for systems that are going to interact with users, since in this case latency is critical and cloud systems are built for low-latency web applications.

Another common place: **edge devices**. Inference can run on your laptop, smartphone, or tablet.

- Here, the size of the model becomes more of an issue: you don't want to fill up your cell phone's entire memory with a neural network!

Running inference on the user's own device is **great for privacy and security**...for the user. It's not so great for the companies that own the models and would like to keep those private.

An emerging place for inference: **IoT devices and/or sensors**. Sometimes we want inference right at the source: on the sensor where data is collected. We can expect this to become increasingly popular as more and more sensors are deployed.

- Example: a surveillance camera taking video
  - Don't want to stream the video to the cloud, especially if most of it is not interesting.
  - Power is the main concern in this setting.