Lecture 13: Accelerating DNN Training.

CS4787 — Principles of Large-Scale Machine Learning Systems

Recap. So far in this course we've talked about a lot of methods for making training more efficient. We discussed all of them in the context of general loss functions and for convex loss functions as a special case. Most of these methods are also applied to deep learning. In fact, momentum, minibatching, and adaptive learning rate schemes are all used regularly for training deep neural networks.

There are also methods for accelerating training that are designed specifically for deep neural networks. Today, we're going to talk about some of these methods.

Review: Overfitting and Underfitting.

- *Underfitting* informally means that the training error is high.
- *Overfitting* informally means that the difference between the test error and the training error is high.
- *Capacity* of a model informally refers to the ability of the model to fit a wide range of possible functions.
  - Models with high capacity tend to overfit.
  - Models with low capacity tend to underfit.
- The *representational capacity* of a parameterized class of models informally refers to the extent to which for a wide range of possible functions, some model in the class approximates that function well.
  - Deep neural networks have very high representational capacity.
  - In fact, they're universal approximators.
- The *effective capacity* of a parameterized class of models *given a specific learning algorithm with a specific amount of data* refers to the extent to which for a wide range of possible functions, the model in the class produced by the learning algorithm can approximate that function well.
- For convex optimization probblems (what we've studied so far) all the algorithms we've studied converge to the global optimum, so effective capacity will be equal to representational capacity.
- On the other hand, changing the optimization algorithm for a deep learning task can alter the effective capacity. Even changing the hyperparameters of an algorithm can have this effect.

Take away point: In addition to thinking about how changes to a model or algorithm affect optimization parameters like n, d, and κ, we also need to reason about how these methods interact with the capacity of the model, especially when we're training a model with a non-convex loss.

Demo. Overfitting for SGD on polynomial regression.

Early stopping. When we train a model with high representational capacity, such as a deep neural network, we often observe that the training loss continues to decrease as we run additional epochs, while the validation error starts to increase at some point.

Motivating idea: we can return a model with better validation error than the final model by just returning the model with the best validation error. We can also save resources by ending execution early if no model has been found that improves over the best observed validation error in the past K epochs, for some threshold

K. This strategy is called *early stopping*.

Question: What are the computational benefits of the early stopping method, compared with ordinary SGD? What are the drawbacks?

Very important point: Do not use the test set for early stopping! This is bad methodology.

Batch normalization. Deep neural networks can have problems during training because of higher-order effects that exist because of the composition of many layers. Small changes made to earlier layers can have a major impact on what happens later in the network. This makes training difficult, because we often want to set the step size to be very small to damp out these higher effects, but this also makes training slow. One way to address this is *batch normalization*, which reparameterizes a deep neural network to reduce these effects.

Suppose that we are concerned with a single (scalar) activation in the network, and we are running mini-batch SGD with batch size B. Let $u \in \mathbb{R}^B$ denote the vector of activations of the original network. Batch normalization replaces this u with

$$\mathsf{BN}(u) = \frac{u - \mu}{\sigma}.$$

where at training time

$$\mu = \frac{1}{B}\sum_{b=1}^{B} u_b \quad \text{and} \quad \sigma = \sqrt{\frac{1}{B}\sum_{b=1}^{B}(u_b - \mu)^2}.$$

Explicitly, we're subtracting out the mean of the minibatch, then dividing by the variance. Importantly, we **still backprop through this** as if it were part of our network. At test time, since there is no minibatch, we instead use values of μ and σ computed from the distribution of that activation over the entire training set.

Commonly, to maintain the expressive power of the network, to do batchnorm we add the additional parameters γ and β and replace the activations with

$$\mathsf{BN}(u) = \gamma \cdot \frac{u - \mu}{\sigma} + \beta.$$

We then backprop through the entire network, using all the old parameters and, additionally, the new parameters γ and β for each activation we are using batch normalization for.

Batch normalization can greatly improve the speed at which we can train a neural network, and it has become standard in many applications.