Lecture 17: Hyperparameter Optimization.

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

Spill-over from last time: kernels continued.

So far in class we've talked about learning algorithms, ways to accelerate these learning algorithms, and deep learning. Each one of these methods introduces many **hyperparameters** that we need to set for the algorithms to run efficiently. And we saw that setting the hyperparameters suboptimally can lead to slow convergence!

Question: What hyperparameters have we seen so far in this course? What bad things can happen if we set them inefficiently?

One way to get a feeling for what hyperparameters do is **theory**. We've seen some amount of theory in this course. One thing that theory can do is **give us a recipe for assigning hyperparameters**.

- For example, theory can tell us how to set hyperparameters for strongly convex optimization if we know the strong convexity constant μ and the Lipschitz constant L.
- Setting hyperparameters according to theory generally gives us the right asymptotic rate.
- And it gives us a good sense of what order of magnitude the hyperparameters should probably have, and how they should scale with problem parameters and other hyperparameters.
- We can even (sometimes) get the *optimal* setting of hyperparameters for a given problem class. For example, for convex optimzation, we can find the best setting of hyperparameters that makes the algorithm converge for any strongly convex, Lipschitz continuous function with that μ and L.
- But it can be loose relative to the best setting of the hyperparameters for a particular problem/dataset!

Summary: the theory I showed you is doing something like

$$\arg \min_{\text{hyperparams}} \max_{\text{data}} \text{ objective}.$$

It's finding a good set of hyperparameters that will work no matter what the data is, as long as the assumptions we made are satisfied.

Key point: This theoretical analysis does not use the training data at all to set the parameters. (Other than perhaps using it to calculate parameters like μ and L).

Second key point: Theory only minimizes an upper bound on the objective...but the actual algorithm could do much better than the bound.

Motivating question: Can we **use the data** to improve our choice of parameters?

Hyperparameter Optimization. Any method that chooses hyperparameters automatically can be called hyperparameter optimization. This is also called "metaparameter optimization" or "tuning."

Question: What's the difference between hyperparameter optimization and regular training? To put it another way, what's the difference between the model parameters and the hyperparameters?

Many possible goals of hyperparameter optimization:

- Simplest setting: we just care about the accuracy of the model we are going to learn, so we just want to minimize the validation error.
- When we care about performance of our training process on hardware, we also need to factor in the cost of running training! This is especially important when we want to use hyperparameter optimization to select systems properties like the number of cores used for parallel training.
- When we care about performance of later inference with the learned model, we also need to factor in this cost of inference.

There are many ways to do hyperparameter optimization. Here are a few major categories?

► **The simplest strategy: hyperparameters from folklore.** Idea: just set the hyperparameters by following what people have done in previous work on similar applications. Or, even what you can remember off the top of your head.

Examples?

Tradeoffs of using hyperparameters set from folklore:

- **Folklore can lead you astray.** It's relatively easy to find tasks for which the standard hyperparameter settings are wrong or suboptimal.
- **But folklore is folklore for a reason.** These are hyperparameter settings that people have found empirically get good results. So it's likely that you will also get good results for your application if it's representitive of the type of applications for which others have observed these good results.
- **Main benefit of using hyperparameters from previous work: you don't have to find the hyperparameters yourself!** Minimizes your computational cost.

► **The most complicated strategy: manual hyperparameter tuning.** Idea: just fiddle with the hyperparameters by hand until you either get the results you want or give up. Preferably this should be an expert human, but even non-experts can do good work here. This is probably the most common type of hyperparameter optimization used in practice (and you've done it before yourself in the programming assignments for this course).

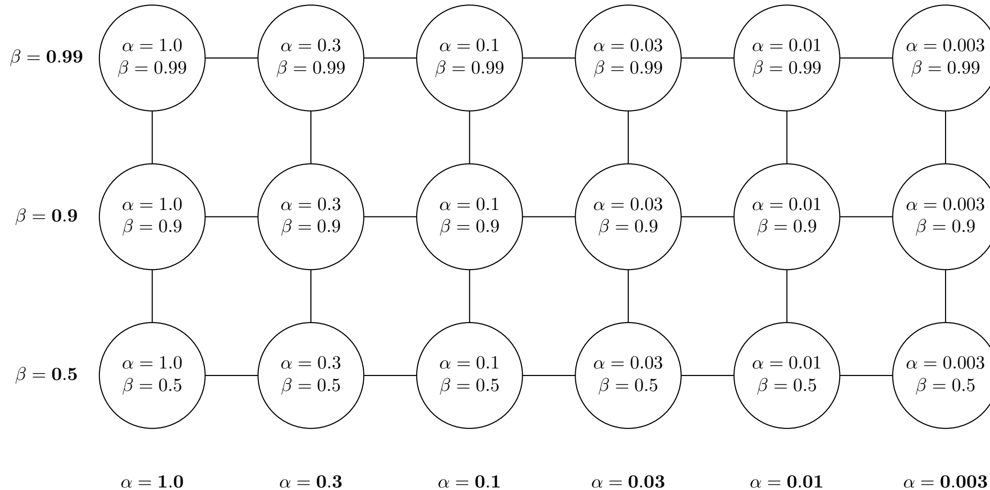
Tradeoffs of tuning by hand:

- **Upside: Results are generally of high quality.** Humans are smart, and expert humans in particular can usually tease out good results if they are there.
- **Subsumes all other types of hyperparameter optimization.** A human tuner can always choose to use any other tuning method as input to their hand-tuning process.

- **There are a lot of humans.** And the vast majority of human time is as yet unused for hyperparameter optimization!
- **Downside: Human effort doesn't scale.** Especially expert human effort.
- **And it's a lot of effort.** Most humans would rather be doing something else.
- **And there's no theoretical guarantees.** Even if there is a good hyperparameter setting, there's no guarantee a particular human researcher will find it.
- **And it's hard to repeat the tuning process on a different dataset or problem.** Makes it difficult to replicate previous work.

► **One automated method: Grid search.** Idea: define some set of values you want to consider for each parameter. Form a grid from the Cartesian product of these sets. Then try all the parameter values in the grid, by running the whole system for each setting of the parameters. Then choose the setting that gives the best results (depending on the metric we care about).

Example of a grid for grid search for momentum SGD:



Tradeoffs of grid search:

- **Upside: Is guaranteed to explore the space.** Or at least it's guaranteed to explore the grid you told it to explore.
- **Highly replicable.** Anyone else can re-run your hyperparameter optimization algorithm easily.
- **Downside: Cost increases exponentially with the number of parameters.** This is the curse of dimensionality.
- **We still need some way to choose the grid properly.** Sometimes this can be almost as hard as the original hyperparameter optimization problem.
- **No way to take advantage of insight you have about the system** or about how the hyperparameters could interact.

How can we make grid search fast? Two simple tricks I'll mention here:

- **Early stopping.** Can abort some of the runs in the grid early if it becomes clear that they will be suboptimal.

- **Parallelism.** Grid search is an embarassingly parallel method. We can evaluate the grid points independently on parallel machines and get near-linear speedup.

Can we use one of our principles of large-scale machine learning to improve grid search? Recall the two major principles we've discussed so far.

▶ Combine grid search with subsampling. **The resulting method: Random search.** Idea: define some distribution over each parameter you want to explore. Take some number N of random samples from the joint distribution of all the parameters. Try all those sampled values, by running the whole system for each setting of the parameters. Then choose the setting that gives the best results.

Tradeoffs of random search vs grid search:

- **Upside: Solves the curse of dimensionality.** We often don't need to increase the number of sample points exponentially as the number of dimensions increases.
- **Downside: Not as replicable.** Results depend on a random seed.
- **Downside: Not necessarily going to go anywhere near the optimal parameters in a finite sample.**
- Same downsides as grid search: needing to choose the search space and no way of taking advantage of insight about the system.

When does random search work better than grid search? A quote:

> " In this work we show that random search is more efficient than grid search in high-dimensional spaces because functions Ψ of interest have a low effective dimensionality; essentially, Ψ of interest are more sensitive to changes in some dimensions than others (Caflisch et al., 1997). In particular, if a function f of two variables could be approximated by another function of one variable ($f(x_1, x_2) \approx g(x_1)$), we could say that f has a low effective dimension.
>
> *"Random Search for Hyper-Parameter Optimization"*, Bergstra and Bengio

▶ Do hyperparameter search with optimization. **One resulting method: Derivative-free optimization (DFO).** Idea: usually in hyperparameter optimization problems we can't automatically differentiate with respect to the hyperparameters (if the objective is even differentiable with respect to the hyperparameters at all). So we can't use powerful large-scale optimization methods like GD and SGD. One class of methods that we can use is called derivative-free optimization methods. These methods solve an optimization problem without using derivative information. Instead they just use objective values (rather than objective gradients) to optimize. Just like SGD and GD, they (usually) operate by iteratively updating a value for the parameters by moving them in a direction that seems to decrease the loss.

Derivative free optimization is generally slow relative to other methods that can be used. (In particular, it's often asymptotically slower than gradient-based methods when they can be used, which is why it isn't usually used for differentiable objective.) But it still sees occasional use for hyperparameter optimization, as well as for cases where we want to minimize a loss that is not differentiable.

Question: how can we incorporate our own knowledge and insights about the problem into an optimization method?

▶ **Bayesian Optimization.** Basically a "smart search." This is what we're going to cover next time!