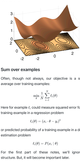
Optimization in ML

As we've seen, a lot of machine learning methods are based on optimizing something: maximizing a likelihood, minimizing an error measure, maximizing information contents etc. Let's say we want to solve

$$\min_{\mathbf{x} \in \mathbb{R}^n} L(\mathbf{x}) \quad \mathbf{x} \in \mathbb{R}^n$$

Here $L(\mathbf{x})$ is our objective, and $\mathbf{x}$ is a d-dimensional parameter vector.

In our examples so far, the objective has often been convex. Convex functions are nice because they have no local minima, but a nearly arbitrary shape, including multiple local optima:



Sum over examples

Often, though not always, our objective is a sum or average over training examples:

$$\min_{\mathbf{x}} \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{x})$$

Here for example, L could measure squared error for a training example in a regression problem:

$$L(\mathbf{x}) = (\mathbf{a} \cdot \mathbf{x} - \mathbf{x}')^2$$

or predicted probability of a training example in a density estimation problem:

$$L(\mathbf{x}) = P(\mathbf{a} \mid \mathbf{x})$$

For the first part of these notes, we'll ignore this structure. But, it will become important later.

The first order

We will focus on first-order methods for optimization, which only assume that we can get a loss function only through its value and gradient.

Even though they're simple, first-order methods turn out to be quite effective for a lot of machine learning problems. This happens for a few different reasons:

- Speed: each optimizer step can be fast. If we have big data or big models, we might not be able to afford fancier optimizers.
- Just enough accuracy: one of the big blockers for machine learning is generalization error, not accuracy. But I will, we often don't care about getting really high accuracy. Those last few decimal points don't matter for the training set anyhow, compared to overfitting, and don't translate to the test set.
- Exploration the trajectory of a first-order method can stop over small increases in the loss, helping us find a better minimum. We'll see this in more detail around the regular new a local optimum, and head on characterize the shape of the objective function there.
- Regularization: For reasons we don't completely understand, first-order methods seem to find a kind of that generalize better than other ways of fitting with the same $L(\mathbf{x})$.

For these reasons, first-order methods are often the best place to start if we need to pick an optimizer for machine learning.

Gradient descent

To solve $\min_{\mathbf{x}} L(\mathbf{x})$ we have already seen the gradient descent algorithm:

Gradient descent

Input $\mathbf{x}_0, \eta, T$
For $t = 1, \ldots, T$:
$\mathbf{g}_t = \nabla L(\mathbf{x}_t)$
$\mathbf{x}_t = \mathbf{x}_t - \eta \mathbf{g}_t$

The vector $\mathbf{g} \in \mathbb{R}^d$ is the gradient of L at $\mathbf{x}_t$, and is sometimes also written ∇L. The parameter η, the learning rate or step size, we'll see below that it trades off between optimization speed and stability.

A lot of times, we won't wind up using gradient descent itself, for reasons to be discussed below. But its cousins, stochastic gradient descent, can often work quite well. Understanding gradient descent is the first step of analysis, so we'll start by looking at plain gradient descent.

Why does gradient descent work? In the neighborhood of $\mathbf{x}_t$ the function L increases most rapidly in the gradient direction $\mathbf{g}_t$:

So, if we take a step in the opposite direction, it should reduce L.

Local quadratic model

Can we make this intuition precise? Let's look at a 1d slice of our objective $L(x)$ by restricting to the line that goes through $\mathbf{x}_t$ and $\mathbf{g}_t$:

We can make a first-order Taylor expansion:

$$L(x) \approx L(x_t) + (x - x_t) \cdot \frac{\partial}{\partial x} L(x_t)$$

Here the residual $R(x)$ is an unknown function — but by the mean value theorem, we know that $R(x)$ is equal to the second derivative of our slice of L, evaluated at some point along the line segment between x_t and x.

Suppose we know that the second derivative is bounded, at least in the neighborhood of x_t. In particular, let's say $|R(x)| \leq H$ for some constant $H \geq 0$. Remembering that we also $R(x_t) = 0$, we know that — say, we get

$$L(\mathbf{x}_t) \leq L(\mathbf{x}) + \mathbf{g}_t(\mathbf{x}_t - \mathbf{x}) + \frac{H}{2}\|\mathbf{x}_t - \mathbf{x}\|^2$$

So, we bring up $\mathbf{x}$ a H and $\mathbf{x} = \mathbf{x} - \eta \mathbf{g}_t$, we get a decrease in L. If we set $\eta = 1/H$, we will reduce L by at least

$$\frac{\|\mathbf{g}_t\|^2}{2H}$$

on each step.

In other words, we make progress as long as we take a small enough step, the best step size depends on the curvature of L. If H is very big, it means that the gradient is far big compared to the local curvature. If we know the curvature, the size of the gradients, and the differences between the initial and final values of L, we can predict how long our optimization will take.

Exercise: what happens to gradient descent if we multiply the objective $\mathbf{R}$ $\mathbf{R}$ by a constant $\mathbf{R}$ $\mathbf{R}$ $\mathbf{R}$

learning rate: one due to the way the curvature of L changes the deterministic gradient (as before) and one due to variance (new for SGD). If we didn't have to worry about variance, our analysis of plain gradient descent shows that we could improve L by $\|\mathbb{E}(g_t)\|^2/2H$ per iteration by setting $\eta = 1/H$. But this setting of η results in a penalty of σ^2/H due to gradient variance. So, if σ^2 is comparable to or bigger than $\|E(g_t)\|^2$, we may have to reduce our learning rate to make progress.

Behavior of SGD

Because of the differing effects of curvature and variance, SGD typically has three different phases of convergence:

- Far away from the optimum, the gradient samples all point in about the same direction. So, $\|E(g_t)\|^2$ dominates, and SGD behaves mostly like plain gradient descent.
- A bit nearer to the optimum, the gradient variance and squared gradient norm become comparable. In this regime our progress depends on moving slowly enough to average out the gradient variance, and SGD will make much less progress per iteration than plain gradient descent. (It may still be faster overall, since the iterations are a lot cheaper.)
- When we get sufficiently close to the optimum, the gradient variance dominates, and we stop reducing L at all. Instead we bounce around at a scale determined by σ^2, H, and η.

We can tune the transitions between these phases by tuning the learning rate: in the range we care about, changing η affects the variance of ηg_t as $O(\eta^2)$, but it affects our reduction in L from the expected update $\mathbb{E}(\eta g_t)$ as $O(\eta)$. So reducing η can mitigate the effect of gradient variance; for example, it can allow the second phase to make more progress, getting us closer to a local optimum before we start bouncing around.

Unfortunately, though, the best value of η might be different during different parts of the overall optimization. For this reason, it's common to alter η during the course of optimization using a *learning rate schedule*. Designing a learning rate schedule is unfortunately somewhat of a black art: at one point, one of my past professors set up his workstation so that his mouse pointer controlled learning rate and momentum, and learned to achieve the fastest possible convergence by printing a lot of diagnostics and developing an intuition for what could keep the network on the ragged edge of stability.

Perhaps counterintuitively, once we fix η, the three phases above *do not depend at all* on the size of our training set: all that matters is the curvature of L and the variance of g_t. SGD might "finish" (i.e., reach the last regime above) before even seeing all of our data.

Instead, the size of our training set influences optimization only indirectly, through our choice of hyperparameters (learning rate schedule and, once we get to it, minibatch size): with a larger training set, we can hope to generalize better. So, we may want to select different hyperparameters to make our optimization more accurate. These altered hyperparameters can change our convergence rate.

Behavior of minibatch SGD

What happens when we use minibatches of size $B > 1$? Broadly we get similar behavior to SGD, but there are three interacting effects we need to consider.

The first effect is computational cost: if we double B, we have to compute twice as many gradients for each parameter update. The second effect is variance: if we double B, we halve the variance of our gradient estimates. The third effect is curvature: the gradient of L changes as we update θ_t, and this change might wipe out any benefit of an accurate gradient estimate.

Looking at the first two effects together, it makes sense to compare:

- two steps of SGD with minibatch size B and learning rate η, vs.
- one step of SGD with minibatch size $2B$ and learning rate 2η.

These approaches seem similar at first look. If the variance of a single gradient sample is bounded by σ^2 in all directions, then either of these approaches yields a total update variance of $2\eta^2\sigma^2/B$. Either of these approaches yields a similar decrease in L, at least to first order. And, either of these approaches has to compute $2B$ gradients in total.

There are some factors that break the apparent equivalence, though. Computing $2B$ gradient samples might take a different amount of wall-clock time depending on whether we do them all at once or split them up. This is particularly true if we can compute the samples in parallel (e.g., on a GPU); in this case the all-at-once approach could be much cheaper.

On the other hand, curvature kicks in if we try to grow B too much. One way to see this is to remember that step size is limited both by the variance of our stochastic gradient estimates and by the effect of the curvature of L on the deterministic gradient direction. For small minibatches, typically the first limit will be tighter; in this case, our two approaches will have similar performance. But bigger minibatches increase the step size while keeping the variance approximately constant. So eventually, the second limit will take over and keep us from increasing B and η too much.

Finally, our minibatch size might be limited by architectural factors. For example, we might want to ensure that each minibatch fits within our GPU memory or our main memory cache.

Empirically, good minibatch sizes can vary greatly depending on details of our loss function and computational architecture. Reasonable values might range anywhere from $B = 10$ to $B = 100,000$.

Over this range of B, we actually have a fair bit of flexibility: we can often achieve good performance with a wide range of different minibatch sizes and learning rates. But there is one important caveat: the analysis above tells us that we have to scale η along with B. If we fail to do this, we can wind up implicitly using a mis-tuned learning rate, and hurting performance.

Suggested reading:

For GD and SGD:

Boyd & Vandenberghe, *Convex Optimization*, sec. 9.3

For an analysis of momentum:

http://mitliagkas.github.io/ift6085-2019/ift-6085-lecture-5-notes.pdf