The zen of gradient descent

Sep 7, 2013 • Moritz Hardt

Ben Recht spoke about optimization a few days ago at the Simons Institute. His talk was a highly entertaining tour de force through about a semester of convex optimization. You should go watch it. It's easy to spend a semester of convex optimization on various guises of gradient descent alone. Simply pick one of the following variants and work through the specifics of the analysis: conjugate, accelerated, projected, conditional, mirrored, stochastic, coordinate, online. This is to name a few. You may also choose various pairs of attributes such as "accelerated coordinate" descent. Many triples are also valid such as "online stochastic mirror" descent. An expert unlike me would know exactly which triples are admissible. You get extra credit when you use "subgradient" instead of "gradient". This is really only the beginning of optimization and it might already seem confusing.

Thankfully, Ben kept things simple. There are indeed simple common patterns underlying many (if not all) variants of gradient descent. Ben did a fantastic job focusing on the basic template without getting bogged down in the details. He also made a high-level point that I strongly agree with. Much research in optimization focuses on convergence rates. That is, how many update steps do we need to minimize the function up to an epsilon error? Often fairly subtle differences in convergence rates are what motivates one particular variant of gradient descent over another. But there are properties of the algorithm that can affect the running time more powerfully than the exact convergence rate. A prime example is robustness. basic gradient descent is robust to noise in several important ways. Accelerated gradient descent is much more brittle. Showing that it is even polynomial time (and under what assumptions) is a rather non-trivial exercise depending on the machine model. I've been saying for a while now that small improvements in running time don't trump major losses in robustness. The situation in optimization is an important place where the trade-off between robustness and efficiency deserves attention. Generally speaking, the question "which algorithm is better" is rarely answered by looking at a single proxy such as "convergence rate".

With that said, let me discuss gradient descent first. Then I will try to motivate why it makes sense to expect an accelerated method and how one might have discovered it. My exposition is not particularly close to Ben's lecture. In particular, mistakes are mine. So, you should still go and watch that lecture. If you already know gradient descent, you can skip/skim the first section.

The basic gradient descent method

The goal is to minimize a convex function $f\colon R^n \to R$ without any constraints. We'll assume that f is twice differentiable and strongly convex. This means that we can squeeze a parabola between the tangent plane at x given by the gradient and the function itself. Formally, for some $\ell > 0$ and all $x, z \in R^n$:

$$f(z) \geq f(x) + \nabla f(x)^T (z - x) + \frac{\ell}{2} \|z - x\|^2.$$

At the same time, we don't want the function to be "too convex". So, we'll require the condition, often called *smoothness*:

$$f(z) \leq f(x) + \nabla f(x)^T (z - x) + \frac{L}{2} \|z - x\|^2.$$

This is a Lipschitz condition on the gradient map in disguise as it is equivalent to:

$$\|\nabla f(x) - \nabla f(z)\| \leq L\|x - z\|.$$

Let's be a bit more concrete and consider from here on the important example of a convex function $f(x) = \frac{1}{2}x^T Ax - b^T x,$ where A is an $n \times n$ positive definite matrix and b is a vector. We have $\nabla f(x) = Ax - b.$ It's an exercise to check that the above conditions boil down to the spectral condition: $\ell I \preceq A \preceq LI.$ Clearly this problem has a unique minimizer given by $x^* = A^{-1} b.$ In other words, if we can minimize this function, we'll know how to solve linear systems.

Now, all that gradient descent does is to compute the sequence of points

$$x_{k+1} = x_k - t_k \nabla f(x_k)$$

for some choice of the step parameter t_k. Our hope is that for some positive $\alpha < 1$,

$$\|x^* - x_{k+1}\| \leq \alpha\|x_k - x^*\|,$$

If this happens in every step, gradient descent converges exponentially fast towards the optimum. This is soberly called linear convergence in optimization. Since the function is smooth, this also guarantees convergence in objective value.

Choosing the right step size t is an important task. If we choose it to small, our progress will be unnecessarily slow. If we choose it too large, we will overshoot. A calculation shows that if we put $t = 2/(\ell + L)$ we get

$\alpha = (L - \ell)/(L + \ell).$ Remember that $\kappa = L/\ell$ is condition number of the matrix. More generally, you could define the condition number of f in this way. We have shown that

$$\|x_k - x^*\| \le \left(\frac{\kappa - 1}{\kappa + 1}\right)^k \|x_0 - x^*\|.$$

So the potential function (or Lyapunov function) drops by a factor of roughly $1 - 1/\kappa$ in every step. This is the convergence rate of gradient descent.

Deriving acceleration through Chebyshev magic

What Nesterov showed in 1983 is that we can improve the convergence rate of gradient descent without using anything more than gradient information at various points of the domain. This is usually when people say something confusing about physics. It's probably helpful to others, but physics metaphors are not my thing. Let me try a different approach. Let's think about why what we were doing above wasn't optimal. Consider the simple example $f(x) = \frac{1}{2}x^T Ax - b^T x.$ Recall, the function is minimized at $A^{-1}b$ and the gradient satisfies $\nabla f(x) = Ax - b.$ Let's start gradient descent at $x_0 = tb.$ We can then check that

$$x_k = \left(\sum_{j=0}^{k} (I - A')^k\right)b'$$

where $A' = tA$ and $b' = tb.$ Why does this converge to $A^{-1}b$? The reason is that what gradient descent is computing is a degree k polynomial approximation of the inverse function. To see this, recall that for all scalars $|x| < 1,$

$$\frac{1}{x} = \sum_{k=0}^{\infty} (1 - x)^k.$$

Since the eigenvalues of A' lie within $(0, 1),$ this scalar function extends to the matrix case. Moreover, the approximation error when truncating the series at degree k is $O((1-x)^k)).$ In the matrix case this translates to error $O(\|(I - A')^k\|) = O((1 - \ell/L)^k).$ This is exactly the convergence rate of gradient descent that we determined earlier.

Why did we go through this exercise? The reason is that now we see that to improve on gradient descent it suffices to find a better low-degree approximation to the scalar function $1/x$. What we'll be able to show is that we can save a square root in the degree while achieving the same error! Anybody familiar with polynomial approximation should have one guess when hearing "quadratic savings in the degree":

Chebyshev polynomials

Let's be clear. Our goal is to find a degree k polynomial$q_k(A)$ which minimizes the residual

$$r_k = \|(I - Aq_k(A))b\|.$$

Put differently we are looking for a polynomial of the form $p_k(z) = 1 - zq(z)$. What we want is that the polynomial is as small as possible on the location of the eigenvalues of A which lie in the interval $[\ell, L]$. At the same time, the polynomial must satisfy $p_k(0) = 1$. This is exactly the property that Chebyshev polynomials achieve with the least possible degree! Quantitatively, we have the following lemma that I learned from Rocco Servedio. As Rocco said in that context:

> *There's only one bullet in the gun. It's called the Chebyshev polynomial.*

Lemma. There is a polynomial p_k of degree $O(\sqrt{(L/\ell)\log(1/\epsilon)})$ such that $p_k(0) = 1$ and $p_k(x) \le \epsilon$ for all $x \in [\ell, L]$.

The lemma implies that we get a quadratic savings in degree. Since we can build p_k from gradient information alone, we now know how to improve the convergence rate of gradient descent. It gets better. The Chebyshev polynomials satisfy a simple recursive definition that defines the k-th degree polynomial in terms of the previous two polynomials. This means that accelerated gradient descent only needs the previous two gradients with suitable coefficients:

$$x_k = x_{k-1} - \alpha_k \nabla f(x_{k-1}) + \beta_k \nabla f(x_{k-2}).$$

Figuring out the best possible coefficients α_k, β_k leads to the above convergence rate. What's amazing is that this trick works for any convex function satisfying our assumptions and not just the special case we dealt with here! In fact, this is what Nesterov showed. I should say that the interpretation in terms of polynomial approximations is lost (as far as I know).The polynomial approximation method I described was known much earlier in the context of eigenvalue computations. This is another fascinating connection I'll describe in the next section.

Let me add that it can be shown that this convergence rate is optimal for any first-order (gradient only method) by taking A to be the Laplacian of a path of length n. This is true even in our special case. It's optimal though in a weak sense: There is a function and a starting point such that the method needs this many steps. I would be interesting to understand how robust this lower bound is.

The connection to eigenvalue methods

Our discussion above was essentially about eigenvalue location. What does polynomial approximation have to do with eigenvalues? Recall, that the most basic way of computing the top eigenvalue of a matrix is the power method. The power method corresponds to a very basic polynomial, namely $p_k(x) = x^k$. This polynomial has the effect that it maps 1 to 1 and moves every number $|x| < 1$ closer to 0 at the rate $|x|^k$. Hence, if the top eigenvalue is λ and the second eigenvalue is $(1-\epsilon)\lambda$, then we need about $k \approx 1/\epsilon$ iterations to approximately find λ. Using exactly the same Chebyshev idea, we can improve this to $k = O(\sqrt{1/\epsilon})$ iterations! This method is often called Lanczos method. So, we have the precise correspondence:

> *The power method is to Lanczos as basic gradient descent is to Accelerated gradient descent!*

I find this quite amazing. In a future post I will return to the power method in greater detail in the context of noise-tolerant eigenvalue computation.

Why don't we teach gradient descent in theory classes?

I'm embarrassed to admit that the first time I saw gradient descent in full generality was in grad school. I had seen the Perceptron algorithm in my last year as an undergraduate. At the time, I was unaware that like so many algorithms it is just a special case of gradient descent. Looking at the typical undergraduate curriculum, it seems like we spend a whole lot of time iterating through dozens of combinatorial algorithms for various problems. So much so that we often don't get around to teaching something as fundamental as gradient descent. It wouldn't take more than two lectures to teach the contents of this blog post (or one lecture if you're Ben Recht). Knowing gradient descent seems quite powerful. It's not only simple and elegant. It's also the algorithmic paradigm behind many algorithms in machine learning, optimization and numerical computation. Teaching it to undergraduates seems like a must. I just now realize that I haven't been an undergraduate in a while. Time flies. So perhaps this is already happening.

More pointers

- Trefethen-Bau, “Numerical Linear Algebra”. My favorite book on the topic of classical numerical methods by far.
- Ben Recht’s lecture notes here and here, his Simonstalk.
- Sebastien Bubeck’s course notes are great!
- For lack of a better reference, the lemma I stated above appears as Claim 5.4 in this paper that I may have co-authored.