CS 542G: Unconstrained Optimization

Robert Bridson

October 22, 2008

1 Optimization

We've already seen several instances of optimization: our first sally into interpolation that ended up with radial basis functions was derived from minimizing a measure of roughness, our approach to dealing with noise in data was to find an optimal fit in the sense of least squares, PCA is based on minimizing the error associated with a rank k approximation of a matrix, and minimization also popped up in algorithms that attempt to find eigenvalues. In these cases, we mostly either had available a full solution based on linear algebra, or a special-purpose algorithm (such as Rayleigh quotient iteration) which performs very well. Let us now turn instead to the general problem of minimizing a function, a function we might know fairly little about. This is relevant to countless applications, but we'll skip over any new motivational problems for now.

The subject of optimization can be split into categories in many ways. Our first decision will be to focus on **unconstrained optimization** where the usual problem is:

$$\arg\min_x f(x)$$

which is often abbreviated to $\min_x f(x)$ even though it's the value of x at a minimum we usually care most about, not $f(x)$. Here x is taken from all of $\mathbb{R}^n$. **Constrained optimization** would restrict the set of allowed x, for example requiring that a constraint function $g(x)$ be zero or non-negative; this considerably complicates the problem, so we'll tackle the simpler version first.

The first concern in a minimization problem is of course if it's well-posed or not. Does the function have a minimum? Is it attained at a unique x? Is that minimizer stable under small perturbations to the data?

The existence of a minimum can be guaranteed by a range of technical conditions on f. For example, if f is continuous and there exists an x^* and a number R such that $f(x) \geq f(x^*)$ whenever $\|x\| > R$,

then $f(x)$ has a minimum. We will assume from now on something along these lines holds true, so there is for sure a minimum. We'll also assume f is at least continuous, if not smoother, since non-smooth optimization problems are virtually intractable unless a lot more is known about the function.

The uniqueness and stability of a minimizer is more often at doubt in practice. Sometimes, for lack of better knowledge or insight in the application, a practitioner will come up with an ill-posed problem with many minimizers or at least an unstable minimizer, which will pose significant difficulties in numerically solving. Sometimes this can be remedied by adding a **regularization** term, perturbing $f(x)$ slightly to make it better behaved. At its simplest, we could minimize $f(x) + \mu\|x\|_2^2$, for some small regularization parameter $\mu > 0$: it's not guaranteed to work, but it could add enough bias towards small-norm x that the problem becomes well-posed while not being too much different than the original.[1] Regularization can be thought of as a best guess stand-in for the *correct* missing terms from f that we don't know.

Even if a minimization problem is well-posed, perhaps after the addition of a suitable regularization term, other challenges can make numerical solution difficult. The main culprit is the presence of multiple local minima: values of x where $f(x)$ is the minimum in a small neighbourhood of x but not globally. Again, without knowing much more about the function, it will be nearly impossible for an algorithm to efficiently determine if a local minimum it finds is actually a global minimum or not. We'll thus rule this case out, or make the assumption that a local minimum is all we need from an algorithm.

2 Convexity

One important and surprisingly large class of optimization problems is the class of **convex** problems. For unconstrained optimization, this means that f is a convex function, meaning its graph never goes above a line segment connecting two points on the graph. More formally, a f is convex if

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y)$$

for any points x and y and any number $\alpha \in [0, 1]$. A **strictly convex** function changes this to a strict inequality when $x \neq y$ and $0 < \alpha < 1$.

Convexity can guarantee many good things for optimization algorithms. For example, a convex function can't have unconnected local minima, and a strictly convex function can only have one local minimum (which must be a global minimum if it exists). The algorithms we will discuss in this course are often generally applicable, but can have special guarantees of convergence or efficiency when used on

[1] We saw a more subtle form of this regularization already in the pseudo-inverse of a matrix, which for a rank-deficient linear system or least squares problem selects the smallest norm solution from the infinitely many available solutions.

convex functions.

Proving that a function is convex isn't always straightforward, but many rules exist to help out. For example, the sum of two convex functions is also convex, and norms are convex (due to the triangle inequality). We won't go into detail here: look for Boyd and Bandenberghe's book "Convex Optimization" (now available for free download) for a great start on this topic, along with theory, applications and algorithms for convex problems.

3 Classes of Solvers

Another way to start slicing up the field into categories is looking at what information about f is available to algorithms:

1. *Evaluation only*: we can evaluate $f(x)$ for a specified x, but that's all.
2. *Gradients also*: we can also evaluate the gradient $\nabla f(x)$ (now assuming f is differentiable).
3. *Hessians also*: we can also evaluate the Hessian matrix $H(x)$, i.e. all second order partial derivatives of f (now assuming f is twice differentiable).

Even higher order information could in principle be available, but particularly in higher dimensions isn't deemed useful.

4 Evaluation-Based Solvers

This class includes some old favourites such as variations on stochastic search, simulated annealing, and genetic algorithms, though these are more important typically in discrete optimization where x is constrained to only have discrete values (such as only having integer components).

Another example you may have heard of is **Cyclic Coordinate Descent** (CCD). After giving CCD an initial guess at x, the first step is to vary just the first component of x, approximately minimizing $f(x_1, x_2, \ldots, x_n)$ as a 1D function of x_1 alone. The second step then varies just the second component, the third step the third component and so on. After the n'th step, we cycle back to varying the first coordinate, and hope that eventually the iteration will converge to a local minimum. There are unfortunately some cases where plain CCD will fail to converge on a well-posed problem, so it's not generally recommended.[2]

[2]Some simple enhancements to CCD can make it a much more reliable algorithm, however: for example, after taking n steps

We'll delay discussion of how one might solve even these simpler 1D approximate minimization steps to the section below on "line search".

Another interesting member of this class of which you might not have heard is **Pattern Search**. Here a "pattern" of directions $\{d_i\}_{i=1}^m$, a small set of unit length vectors, is first chosen with the property that every nonzero vector in $\mathbb{R}^n$ will have a strictly positive dot-product with at least one of the directions. An initial step size α is chosen to begin, with an initial guess $x^{(0)}$. At guess $x^{(k)}$, f is evaluated at $x^{(k)} + \alpha d_i$ for each direction d_i (a kind of star pattern around the current guess); if any value is less than the lowest value seen so far, that becomes the new guess $x^{(k+1)}$. Otherwise the step size is halved $\alpha \leftarrow \frac{1}{2}\alpha$ and the directions are tried again. If the step size is reduced too many times without finding a new minimum, the algorithm is deemed to converge; if a given step size succeeds too many times in a row, a larger step like $\frac{3}{2}\alpha$ can be tried to accelerate the algorithm. This is guaranteed to converge to a local minimum for a well-posed problem, but it certainly can be slow.

This is the main complaint about this category: though some algorithms in it are reliable, it's unlikely that any will be particularly fast, and for challenging problems f may be too slow to be practical (what we mean by challenging will be elucidated next lecture). However, if f isn't smooth enough (perhaps it has a degree of noise inherent in it) or is just too complex to differentiate practically (say it's actually a ten million line legacy simulation code for evaluating the efficiency of an aircraft design, and we're trying to optimize that design), this might be the only category available.

5 Gradient Solvers

The second category, where we assume f is differentiable (and probably a bit more, such as being at least continuously differentiable), is much richer. The classic example of this category is **Steepest Descent**, which we'll spend some time with as it forms a basis for almost all the more sophisticated methods, which do include the **quasi-Newton** solvers which also lie in this category.

5.1 Steepest Descent

Recall the **gradient** of a function, i.e. the vector of all its first partial derivatives

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

(one along each coordinate direction) the difference between the first of those steps and the last step can be taken as an extra direction to search along. See the section on line search for a little more detail.

from multivariate calculus. One of its properties is in evaluating the **directional derivative** of a function, which given some direction vector u (typically unit-length) can be defined as the regular first derivative along this direction:

$$\frac{\partial f}{\partial u}(x) = \lim_{h \to 0} \frac{f(x + hu) - f(x)}{h}$$

For reasonable f this is equal to $\nabla f(x) \cdot u$; clearly then ∇f points in the direction of steepest increase (or **steepest ascent**) of the function, and more importantly for minimization, $-\nabla f(x)$ points in the direction of **steepest descent**.

The **Steepest Descent** algorithm, at each iteration, evaluates this direction as apparently the most promising—at least in a "greedy" sense—and seeks to approximately minimize $f(x)$ restricted to this 1D line to get the next guess:

- Begin with initial guess $x^{(0)}$
- For $k = 1, 2, \ldots$ until convergence:
  - Evaluate the **search direction** $d = -\nabla f(x)$
  - Find an $\alpha > 0$ that approximately minimizes $f(x^{(k-1)} + \alpha d)$
  - Set $x^{(k)} = x^{(k-1)} + \alpha u$

The two big questions that this simple algorithm leave are how to find an α in the 1D search, and how to check when we are converged.

6 Line Search

The subject of minimizing a 1D function—usually in the context of finding a **step size** α to minimize $f(x+\alpha d)$ for some previous guess x and direction d—is called **line search**. It's not just relevant to steepest descent, but also the CCD algorithm mentioned above, and many other more sophisticated algorithms as well.

While this still isn't the easiest problem to solve fully, typically we only need an α that's "good enough", reducing the value of f enough that progress is made in converging to the full solution. This isn't entirely easy to define, or derive an effective algorithm that can guarantee it, but the following **backtracking** line search is a great start if you don't want to follow up on more sophisticated line search methods:

- Start with a user-given α
- If $f(x + \alpha d) < f(x)$ then stop.

- Otherwise, try again with $\alpha \leftarrow \frac{1}{2}\alpha$

Note that this should terminate with a positive α as long as the directional derivative of f along d is negative—i.e. d is a **descent direction**, one with $\nabla f(x) \cdot d < 0$—and this is certainly the case for steepest descent. Of course, with rounding errors in play this isn't a strict guarantee and a maximum bound on this loop (after which the algorithm might stop and report it has gone as far as it can) is a good idea.

The very first α is best given by the user, if they have any clue as to what an appropriate ball-park figure for α should be, since without more information about f there's really very little we can do to estimate it *a priori*. However, the last successful α from one step of steepest descent can be used as a guess for the next step, so continual input is not needed. To help speed up convergence if the initial α was overly small, you might actually try increasing it to $\frac{3}{2}\alpha$ at the start of a new line search, which lets it grow exponentially until it hits the right range.

7 Testing for Convergence

A sharp reader may have noticed one other caveat I missed in the section above: if $\nabla f = 0$, so the directional derivative is zero, then line search may well fail. Of course, the sharp reader will also realize this is exactly the condition that x is already at a local minimum. This leads to the simplest way of testing for convergence: is $\|\nabla f(x)\|$ small enough?

Ah, but what should "small enough" be exactly? Ideally we'd pick an ϵ so that $\|\nabla f(x)\| < \epsilon$ implies x is within some desired error margin of the true minimizer, or $f(x)$ is within a desired error margin of the true minimum.

Unfortunately here we also hit the problem of just not having enough information about $f(x)$ to automatically determine a sensible convergence tolerance. A sample of the magnitude of x, of $f(x)$, and/or of $\nabla f(x)$ doesn't tell us what the ϵ might be: observe that minimizing $f(x)$ is more or less equivalent to minimizing $Af(Bx + C) + D$ for $A > 0$ and $B \neq 0$, and this transformation can arbitraily change all of x, $f(x)$, and $\nabla f(x)$ near a minimum.