Overview of Quasi-Newton optimization methods

About this document

These notes were prepared by Galen Andrew for an informal tutorial at Microsoft Research, on Jan. 18, 2008. They are essentially a condensation of the relevent sections of Chapters 6 and 7 from "Numerical Optimization, Second Edition" by Jorge Nocedal and Stephen J. Wright, Springer Verlag, 2006.

Introduction

Suppose we are given a convex,[1] twice-differentiable function to optimize: $f: \mathbb{R}^n \to \mathbb{R}$ without constraints. The methods we will discuss are iterative, starting with an initial point x_0, and producing a sequence of points x_k that converges to the optimum x^*. Denote the gradient of f at x_k by ∇f_k, and the Hessian matrix (the matrix of second partial derivatives) by B_k.

Newton's method

In Newton's method, we find the new iterate x_{k+1} as a function of x_k as follows. For any point x define $p = x - x_k$. The second order Taylor expansion around x_k is given by

$$m_k(p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T \mathrm{B_k} p$$

This defines a *quadratic model* of the function near the point x_k. Its gradient with respect to x is $\nabla m_k(p) = \nabla f_k + B_k p$, and it is minimized at $p_k = -B_k^{-1} \nabla f_k$. (Here we use the assumption that f is convex, so B_k is positive definite.) Finding p_k thus requires computing the inverse of the Hessian, or at least solving the linear system $B_k p = -\nabla f_k$ by other means.

The next point x_{k+1} is then found via a *line-search* in the direction of p_k: for some $\alpha \in (0, \infty)$, $x_{k+1} = x_k + \alpha p_k$. I won't go into detail about the line search, but a good line search (in particular, one that satisfies the Wolfe conditions) is necessary to guarantee convergence. I will note that for these methods, the line search can be very approximate; in most cases the first step size $\alpha = 1$ will be used.

DFP

The first quasi-Newton method is DFP, named after Davidson, who discovered it in 1959, and Fletcher and Powell, who explored its mathematical properties over the next few years. Instead of computing the true Hessian as in Newton's method, we will use an approximation that is based on the change in gradient between iterations. The primary advantage is that we don't have to compute the exact Hessian at each point, which may be computationally expensive.

The idea is to characterize the approximate Hessian B via several properties, and then derive an expression for the unique B that satisfies the properties. The three properties are:

[1] The convexity assumption can be relaxed, in which case these methods can be implemented in such a way that they are still guaranteed to converge to a local minimum. See N&R for details.

1. B_k must be symmetric.
2. When we form a quadratic model using B_k, as above, the gradient of the model must equal the function's gradient at the points x_k and x_{k-1}. This holds trivially for x_k (because $p = 0$ for $x = x_k$). For x_{k-1} the condition means that $\nabla f_k + B_k(x_{k-1} - x_k) = \nabla f_{k-1}$, which we can rewrite as $B_k(x_k - x_{k-1}) = (\nabla f_k - \nabla f_{k-1})$. Defining the *displacement vector* $s_{k-1} = x_k - x_{k-1}$ and the *change in gradient* $y_{k-1} = \nabla f_k - \nabla f_{k-1}$, we can express this as $B_k s_{k-1} = y_{k-1}$, which is known as the *secant equation*.
3. Subject to the above, B_k should be as close as possible to B_{k-1}. The definition of closeness we will use is the *weighted Frobenius norm* of the difference between the Hessians:
$$\|B - B_{k-1}\|_W = \left\|W^{\frac{1}{2}}(B - B_{k-1})W^{\frac{1}{2}}\right\|,$$
where W is any matrix satisfying $W y_{k-1} = s_{k-1}$. This definition of distance ensures that the solution is *non-dimensional, i.e.* it does not change with an arbitrary scaling of the variables.

The second and third properties encode the assuption that the Hessian does not change wildly from iteration to iteration. This assumption may be suspect for the first few iterations, but as the process converges to a point (as all of these algorithms are guaranteed to do), it is not unreasonable. These three properties give us the following optimization problem:

$$\text{minimize } \|B - B_{k-1}\|_W \text{ subject to } B = B^T,\ Bs_{k-1} = y_{k-1}.$$

We set B_k to the unique solution to this problem, which is given by

$$B_k = \left(I - \rho_{k-1} y_{k-1} s_{k-1}^T\right) B_{k-1} \left(I - \rho_{k-1} s_{k-1} y_{k-1}^T\right) + y_{k-1}\rho_{k-1} y_{k-1}^T,$$

where $\rho_k = \left(y_k^T s_k\right)^{-1}$.

Note that we did not require that B_k be positive definite. That is because we can show that it *must* be positive definite if B_{k-1} is. Therefore, as long as the initial Hessian approximation B_0 is positive definite, all B_k are, by induction. We need to show that, if $B_{k-1} \succ 0$, then for any non-zero vector z, $z^T B_k z > 0$. Form the vector $w = z - \rho_{k-1} s_{k-1}(y_{k-1}^T z)$, so that

$$z^T B_k z = w^T B_{k-1} w + \rho_{k-1}\left(y_{k-1}^T z\right)^2.$$

It is easy to show that ρ_k is always positive for any convex function, so this expression is always non-negative. It could be zero only if $y_{k-1}^T z$ is zero, in which case $w = z \neq 0$, so $w^T B_{k-1} w > 0$ by the positive definiteness of B_{k-1}. Therefore $z^T B_k z > 0$, and B_k is positive definite.

The final issue to address for DFP is how to choose the initial Hessian B_0. One option is to use the true Hessian at the initial point (so we still have to compute it once), or a diagonal approximation to the true Hessian. An easier option that works fine in practice is to use a scalar multiple of the identity matrix, where the scaling factor is chosen to be in the range of the eigenvalues of the true Hessian. See N&R for a recipe to find this initializer.

BFGS

DFP was the first quasi-Newton method, but it was soon superceded by BFGS, which is considered to be the most effective quasi-Newton method. BFGS is named for the four people who (independently!) discovered it in 1970: Broyden, Fletcher, Goldfarb and Shanno. It is actually the same as DFP with a single, very elegant modification: instead of approximating the Hessian, B_k, we approximate its inverse H_k, using exactly the same criteria as before. Since the search direction $p = -H_k \nabla f_k$, this has the advantage that we don't need to solve a linear system to get the search direction, but only do a matrix/vector multiply. It is also more numerically stable, and has very effective "self-correcting properties" not shared by DFP, which may account for its superior performance in practice.

Working with the inverse Hessian H_k in place of B_k, the secant equation becomes $H_k y_{k-1} = s_{k-1}$. The optimization is then

$$\text{minimize } \|H - H_{k-1}\|_W \text{ subject to } H = H^T, H y_{k-1} = s_{k-1}.$$

which has the unique solution

$$H_k = \left(I - \rho_{k-1} s_{k-1} y_{k-1}^T\right) H_{k-1} \left(I - \rho_{k-1} y_{k-1} s_{k-1}^T\right) + s_{k-1} \rho_{k-1} s_{k-1}^T.$$

Note the symmetry between the solutions of DFP and BFGS: they are identical except that y and s have been reversed! Therefore, the discussion of positive definiteness and initial approximation is just the same as with DFP.

L-BFGS

BFGS is a very effective optimization algorithm that does not require computing the exact Hessian, or finding any matrix inverses. However, it is not possible to use BFGS on problems with a very high number n of variables (millions, say), because in that case it is impossible to store or manipulate the approximate inverse Hessian H, which is of size n^2. L-BFGS ("limited-memory" BFGS) solves this problem by storing the approximate Hessian in a compressed form that requires storing only a constant multiple of vectors of length n. In particular, L-BFGS only "remembers" updates from the last m iterations, so information about iterates before that is lost. Furthermore, the search direction can be computed in a number of operations that is also linear in n (and m).

The way this is accomplished is by "unrolling" the Hessian update from BFGS as follows. Defining $V_k = I - \rho_k y_k s_k^T$, the BFGS update can be written

$$H_k = V_{k-1}^T H_{k-1} V_{k-1} + s_{k-1} \rho_{k-1} s_{k-1}^T$$

Rolling back one step gives

$$H_k = V_{k-1}^T V_{k-2}^T H_{k-2} V_{k-2} V_{k-1} + V_{k-1}^T s_{k-2} \rho_{k-2} s_{k-2}^T V_{k-1} + s_{k-1} \rho_{k-1} s_{k-1}^T$$

(From now on I will write the subscript $-i$ in place of $k - i$, and it is to be understood that all subscripts are relative to k.) Then we can unroll the computation back m steps:

$$
\begin{aligned}
H_k = {} & (V_{-1}^T V_{-2}^T \dots V_{-m}^T) H_{-m} (V_{-m} V_{-m+1} \dots V_{-1}) \\
& + (V_{-1}^T V_{-2}^T \dots V_{-m+1}^T) s_{-m} \rho_{-m} s_{-m}^T (V_{-m} V_{-m+1} \dots V_{-1}) \\
& + (V_{-1}^T V_{-2}^T \dots V_{-m+2}^T) s_{-m+1} \rho_{-m+1} s_{-m+1}^T (V_{-m+1} V_{-m+2} \dots V_{-1}) \\
& + \cdots \\
& + V_{-1}^T s_{-2} \rho_{-2} s_{-2}^T V_{-1} \\
& + s_{-1} \rho_{-1} s_{-1}^T
\end{aligned}
$$

Remember that ultimately we need to compute the search direction $p_k = -H_k \nabla f_k$. In a moment I will show how to use this recursion to compute p_k without actually expanding out the matrix H_k. All that is required is that we can compute $H_{-m} \nabla f_k$ efficiently. In L-BFGS, H_{-m} plays the role of H_0 in BFGS. It is exactly as if we started at the point x_{-m} and ran m iterations of BFGS, starting with the initial approximation H_{-m}. If H_{-m} has a simple form, like a multiple of the identity matrix, then we can easily compute $H_{-m} \nabla f_k$. Note that unlike the H_0 of BFGS, H_{-m} is allowed to vary from iteration to iteration. For example, we might use a multiple of the identity based on the most recent change in the gradient.

So, how do we compute $H_k \nabla f_k$? With the following algorithm.

```
q = ∇f_k
for (i = 1...m) do
    α_i = ρ_{-i} s_{-i}^T q
    q = q − α_i y_{-i}
end for
r = H_{-m} q
for (i = m...1) do
    β = ρ_{-i} y_{-i}^T r
    r = r + s_{-i}(α_i − β)
end for
return r
```

To see why this works, let q_i be the value of q after iteration i of the first loop. Then $q_0 = \nabla f_k$, and

$$q_i = q_{i-1} - \rho_{-i} y_{-i} s_{-i}^T q_{i-1} = V_{-i} q_{i-1} = (V_{-i} \dots V_{-1}) \nabla f_k,$$

and therefore

$$\alpha_i = \rho_{-i} s_{-i}^T q_{i-1} = \rho_{-i} s_{-i}^T (V_{-i} \dots V_{-1}) \nabla f_k.$$

You can see that the α's are building up the right sides of the unrolled BFGS update. Moving to the r's, we have $r_{m+1} = H_{-m} q = H_{-m} (V_{-i} \dots V_{-1}) \nabla f_k$, and

$$r_i = r_{i+1} + s_{-i} \left(\alpha_i - \rho_{-i} y_{-i}^T r_{i+1} \right) = \left(I - s_{-i} \rho_{-i} y_{-i}^T \right) r_{i+1} + s_{-i} \alpha_i = V_{-i}^T r_{i+1} + s_{-i} \alpha_i.$$

Finally, if we "unroll" this expression for r_1, we get

$$\begin{aligned} r_1 = V_{-1}^T r_2 + s_{-1}\rho_{-1}s_{-1}^T\nabla f_k = (V_{-1}^T V_{-2}^T)r_3 + V_{-1}^T s_{-2}\rho_{-2}s_{-2}^T V_{-1}\nabla f_k + s_{-1}\rho_{-1}s_{-1}^T\nabla f_k = \cdots \\ = (V_{-1}^T V_{-2}^T \dots V_{-m}^T)H_{-m}(V_{-m}V_{-m+1}\dots V_{-1})\nabla f_k + \cdots + V_{-1}^T s_{-2}\rho_{-2}s_{-2}^T V_{-1}\nabla f_k \\ + s_{-1}\rho_{-1}s_{-1}^T\nabla f_k \end{aligned}$$

which is clearly building up the product of our expression for H_k with the gradient ∇f_k.

Note that this process requires only storing s_{-i} and y_{-i} for $i = 1 \dots m$, or $O(mn)$ floating-point values, which may represent drastic savings over $O(n^2)$ values to store H. After each iteration, we can discard the vectors from iteration $k - m$, and add new vectors for iteration k. Also note that the time required to compute p is only $O(mn)$.