


[BLOG \(/blog\)](/blog/) [ACADEMIC \(/academic\)](/academic/) [MEDIA \(/media\)](/media/)

DECEMBER 02, 2014

## Numerical Optimization: Understanding L-BFGS (/blog/2014/12/understanding-lbfgs)

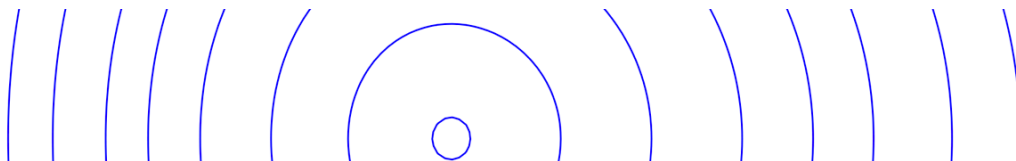
Numerical optimization is at the core of much of machine learning. Once you've defined your model and have a dataset ready, estimating the parameters of your model typically boils down to minimizing some multivariate function ([http://en.wikipedia.org/wiki/Multivariable\\_calculus](http://en.wikipedia.org/wiki/Multivariable_calculus))  $f(x)$ , where the input  $x$  is in some high-dimensional space and corresponds to model parameters. In other words, if you solve:

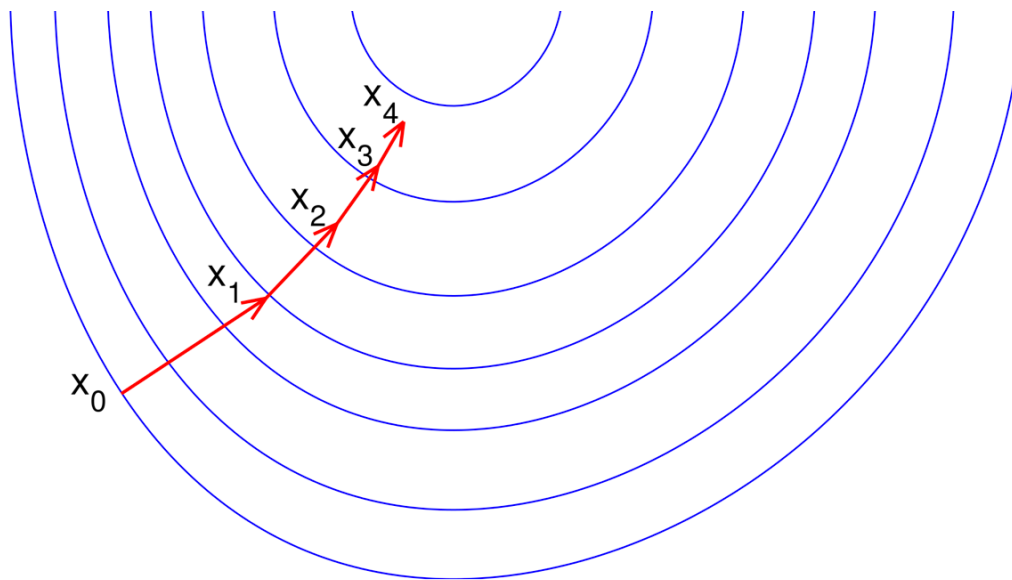
$$x^* = \arg \min_x f(x)$$

then  $x^*$  is the 'best' choice for model parameters according to how you've set your objective.<sup>1</sup>

In this post, I'll focus on the motivation for the L-BFGS ([http://en.wikipedia.org/wiki/Limited-memory\\_BFGS](http://en.wikipedia.org/wiki/Limited-memory_BFGS)) algorithm for unconstrained function minimization, which is very popular for ML problems where 'batch' optimization makes sense. For larger problems, online methods based around stochastic gradient descent ([http://en.wikipedia.org/wiki/Stochastic\\_gradient\\_descent](http://en.wikipedia.org/wiki/Stochastic_gradient_descent)) have gained popularity, since they require fewer passes over data to converge. In a later post, I might cover some of these techniques, including my personal favorite AdaDelta (<http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf>).

**Note:** Throughout the post, I'll assume you remember multivariable calculus. So if you don't recall what a gradient (<http://en.wikipedia.org/wiki/Gradient>) or Hessian ([http://en.wikipedia.org/wiki/Hessian\\_matrix](http://en.wikipedia.org/wiki/Hessian_matrix)) is, you'll want to bone up first.





## Newton's Method

Most numerical optimization procedures are iterative algorithms which consider a sequence of 'guesses'  $x_n$  which ultimately converge to  $x^*$  the true global minimizer of  $f$ . Suppose, we have an estimate  $x_n$  and we want our next estimate  $x_{n+1}$  to have the property that  $f(x_{n+1}) < f(x_n)$ .

Newton's method is centered around a quadratic approximation of  $f$  for points near  $x_n$ . Assuming that  $f$  is twice-differentiable, we can use a quadratic approximation of  $f$  for points 'near' a fixed point  $x$  using a [Taylor expansion](http://en.wikipedia.org/wiki/Taylor_series) ([http://en.wikipedia.org/wiki/Taylor\\_series](http://en.wikipedia.org/wiki/Taylor_series)):

$$f(x + \Delta x) \approx f(x) + \Delta x^T \nabla f(x) + \frac{1}{2} \Delta x^T (\nabla^2 f(x)) \Delta x$$

where  $\nabla f(x)$  and  $\nabla^2 f(x)$  are the gradient and Hessian of  $f$  at the point  $x_n$ . This approximation holds in the limit as  $\|\Delta x\| \rightarrow 0$ . This is a generalization of the single-dimensional Taylor polynomial expansion you might remember from Calculus.

In order to simplify much of the notation, we're going to think of our iterative algorithm of producing a sequence of such quadratic approximations  $h_n$ . Without loss of generality, we can write  $x_{n+1} = x_n + \Delta x$  and re-write the above equation,

$$h_n(\Delta x) = f(x_n) + \Delta x^T \mathbf{g}_n + \frac{1}{2} \Delta x^T \mathbf{H}_n \Delta x$$

where  $\mathbf{g}_n$  and  $\mathbf{H}_n$  represent the gradient and Hessian of  $f$  at  $x_n$ .

We want to choose  $\Delta x$  to minimize this local quadratic approximation of  $f$  at  $x_n$ .

Differentiating with respect to  $\Delta x$  above yields:

$$\frac{\partial h_n(\Delta x)}{\partial \Delta x} = \mathbf{g}_n + \mathbf{H}_n \Delta x$$

Recall that any  $\Delta x$  which yields  $\frac{\partial h_n(\Delta x)}{\partial \Delta x} = 0$  is a local extrema of  $h_n(\cdot)$ . If we assume that  $\mathbf{H}_n$  is [postive definite] (psd) then we know this  $\Delta x$  is also a global minimum for  $h_n(\cdot)$ .

Solving for  $\Delta x$ :

$$\Delta x = -\mathbf{H}_n^{-1} \mathbf{g}_n$$

This suggests  $\mathbf{H}_n^{-1} \mathbf{g}_n$  as a good direction to move  $x_n$  towards. In practice, we set  $x_{n+1} = x_n - \alpha(\mathbf{H}_n^{-1} \mathbf{g}_n)$  for a value of  $\alpha$  where  $f(x_{n+1})$  is 'sufficiently' smaller than  $f(x_n)$ .

## Iterative Algorithm

The above suggests an iterative algorithm:

```

NewtonRaphson( $f, x_0$ ) :
  For  $n = 0, 1, \dots$  (until converged) :
    Compute  $\mathbf{g}_n$  and  $\mathbf{H}_n^{-1}$  for  $x_n$ 
     $d = \mathbf{H}_n^{-1} \mathbf{g}_n$ 
     $\alpha = \min_{\alpha \geq 0} f(x_n - \alpha d)$ 
     $x_{n+1} \leftarrow x_n - \alpha d$ 

```

The computation of the  $\alpha$  step-size can use any number of [line search](http://en.wikipedia.org/wiki/Line_search) ([http://en.wikipedia.org/wiki/Line\\_search](http://en.wikipedia.org/wiki/Line_search)) algorithms. The simplest of these is [backtracking line search](http://en.wikipedia.org/wiki/Backtracking_line_search) ([http://en.wikipedia.org/wiki/Backtracking\\_line\\_search](http://en.wikipedia.org/wiki/Backtracking_line_search)), where you simply try smaller and smaller values of  $\alpha$  until the function value is 'small enough'.

In terms of software engineering, we can treat **NewtonRaphson** as a blackbox for any twice-differentiable function which satisfies the Java interface:

```

public interface TwiceDifferentiableFunction {
    // compute  $f(x)$ 

```

```

public double valueAt(double[] x);

// compute grad f(x)

public double[] gradientAt(double[] x);

// compute inverse hessian H^-1

public double[][] inverseHessian(double[] x);
}

```

With quite a bit of tedious math, you can prove that for a convex function ([http://en.wikipedia.org/wiki/Convex\\_function](http://en.wikipedia.org/wiki/Convex_function)), the above procedure will converge to a unique global minimizer  $x^*$ , regardless of the choice of  $x_0$ . For non-convex functions that arise in ML (almost all latent variable models or deep nets), the procedure still works but is only guaranteed to converge to a local minimum. In practice, for non-convex optimization, users need to pay more attention to initialization and other algorithm details.

## Huge Hessians

The central issue with **NewtonRaphson** is that we need to be able to compute the inverse Hessian matrix.<sup>3</sup> Note that for ML applications, the dimensionality of the input to  $f$  typically corresponds to model parameters. It's not unusual to have hundreds of millions of parameters or in some vision applications even billions of parameters

([http://static.googleusercontent.com/media/research.google.com/en/us/archive/large\\_deep\\_networks\\_nips2012.pdf](http://static.googleusercontent.com/media/research.google.com/en/us/archive/large_deep_networks_nips2012.pdf)).

For these reasons, computing the hessian or its inverse is often impractical. For many functions, the hessian may not even be analytically computable, let alone representable.

Because of these reasons, **NewtonRaphson** is rarely used in practice to optimize functions corresponding to large problems. Luckily, the above algorithm can still work even if  $\mathbf{H}_n^{-1}$  doesn't correspond to the exact inverse hessian at  $x_n$ , but is instead a good approximation.

## Quasi-Newton

Suppose that instead of requiring  $\mathbf{H}_n^{-1}$  be the inverse hessian at  $x_n$ , we think of it as an approximation of this information. We can generalize **NewtonRaphson** to take a QuasiUpdate policy which is responsible for producing a sequence of  $\mathbf{H}_n^{-1}$ .

```

QuasiNewton( $f, x_0, \mathbf{H}_0^{-1}, \text{QuasiUpdate}$ ) :
  For  $n = 0, 1, \dots$  (until converged) :
    // Compute search direction and step-size
     $d = \mathbf{H}_n^{-1} \mathbf{g}_n$ 
     $\alpha \leftarrow \min_{\alpha \geq 0} f(x_n - \alpha d)$ 
     $x_{n+1} \leftarrow x_n - \alpha d$ 
    // Store the input and gradient deltas
     $\mathbf{g}_{n+1} \leftarrow \nabla f(x_{n+1})$ 
     $s_{n+1} \leftarrow x_{n+1} - x_n$ 
     $y_{n+1} \leftarrow \mathbf{g}_{n+1} - \mathbf{g}_n$ 
    // Update inverse hessian
     $\mathbf{H}_{n+1}^{-1} \leftarrow \text{QuasiUpdate}(\mathbf{H}_n^{-1}, s_{n+1}, y_{n+1})$ 

```

We've assumed that **QuasiUpdate** only requires the former inverse hessian estimate as well as the input and gradient differences ( $s_n$  and  $y_n$  respectively). Note that if **QuasiUpdate** just returns  $\nabla^2 f(x_{n+1})$ , we recover exact NewtonRaphson.

In terms of software, we can blackbox optimize an arbitrary differentiable function (with no need to be able to compute a second derivative) using **QuasiNewton** assuming we get a quasi-newton approximation update policy. In Java this might look like this,

```

public interface DifferentiableFunction {
    // compute f(x)

    public double valueAt(double[] x);

    // compute grad f(x)

    public double[] gradientAt(double[] x);
}

public interface QuasiNewtonApproximation {
    // update the H^{-1} estimate (using x_{n+1}-x_n and grad_{n+1}-grad_n)

    public void update(double[] deltaX, double[] deltaGrad);

    // H^{-1} (direction) using the current H^{-1} estimate

    public double[] inverseHessianMultiply(double[] direction);
}

```

Note that the only use we have of the hessian is via it's product with the gradient direction. This will become useful for the L-BFGS algorithm described below, since we don't need to represent the Hessian approximation in memory. If you want to see these abstractions in action, here's a link to a [java 8 \(https://github.com/aria42/java8-optimize/tree/master/src/optimize\)](https://github.com/aria42/java8-optimize/tree/master/src/optimize) and [golang \(https://github.com/aria42/taskar/blob/master/optimize/newton.go\)](https://github.com/aria42/taskar/blob/master/optimize/newton.go) implementation I've written.

## Behave like a Hessian

What form should QuasiUpdate take? Well, if we have QuasiUpdate always return the identity matrix (ignoring its inputs), then this corresponds to simple [gradient descent](http://en.wikipedia.org/wiki/Gradient_descent) ([http://en.wikipedia.org/wiki/Gradient\\_descent](http://en.wikipedia.org/wiki/Gradient_descent)), since the search direction is always  $\nabla f_n$ . While this actually yields a valid procedure which will converge to  $x^*$  for convex  $f$ , intuitively this choice of QuasiUpdate isn't attempting to capture second-order information about  $f$ .

Let's think about our choice of  $\mathbf{H}_n$  as an approximation for  $f$  near  $x_n$ :

$$h_n(d) = f(x_n) + d^T \mathbf{g}_n + \frac{1}{2} d^T \mathbf{H}_n d$$

## Secant Condition

A good property for  $h_n(d)$  is that its gradient agrees with  $f$  at  $x_n$  and  $x_{n-1}$ . In other words, we'd like to ensure:

$$\begin{aligned}\nabla h_n(x_n) &= \mathbf{g}_n \\ \nabla h_n(x_{n-1}) &= \mathbf{g}_{n-1}\end{aligned}$$

Using both of the equations above:

$$\nabla h_n(x_n) - \nabla h_n(x_{n-1}) = \mathbf{g}_n - \mathbf{g}_{n-1}$$

Using the gradient of  $h_{n+1}(\cdot)$  and canceling terms we get

$$\mathbf{H}_n(x_n - x_{n-1}) = (\mathbf{g}_n - \mathbf{g}_{n-1})$$

This yields the so-called "secant conditions" which ensures that  $\mathbf{H}_{n+1}$  behaves like the Hessian at least for the difference  $(x_n - x_{n-1})$ . Assuming  $\mathbf{H}_n$  is invertible (which is true if

it is psd), then multiplying both sides by  $\mathbf{H}_n^{-1}$  yields

$$\mathbf{H}_n^{-1} \mathbf{y}_n = \mathbf{s}_n$$

where  $\mathbf{y}_{n+1}$  is the difference in gradients and  $\mathbf{s}_{n+1}$  is the difference in inputs.

## Symmetric

Recall that the a hessian represents the matrix of 2nd order partial derivatives:

$\mathbf{H}^{(i,j)} = \partial^2 f / \partial x_i \partial x_j$ . The hessian is symmetric since the order of differentiation doesn't matter.

## The BFGS Update

Intuitively, we want  $\mathbf{H}_n$  to satisfy the two conditions above:

- Secant condition holds for  $\mathbf{s}_n$  and  $\mathbf{y}_n$
- $\mathbf{H}_n$  is symmetric

Given the two conditions above, we'd like to take the most conservative change relative to

$\mathbf{H}_{n-1}$ . This is reminiscent of the [MIRA update](#)

(<http://aria42.com/blog/2010/09/classification-with-mira-in-clojure/>), where we have conditions on any good solution but all other things equal, want the 'smallest' change.

$$\begin{aligned} \min_{\mathbf{H}^{-1}} \quad & \|\mathbf{H}^{-1} - \mathbf{H}_{n-1}^{-1}\|^2 \\ \text{s.t.} \quad & \mathbf{H}^{-1} \mathbf{y}_n = \mathbf{s}_n \\ & \mathbf{H}^{-1} \text{ is symmetric} \end{aligned}$$

The norm used here  $\|\cdot\|$  is the [weighted frobenius norm](#)

(<http://mathworld.wolfram.com/FrobeniusNorm.html>).<sup>4</sup> The solution to this optimization problem is given by

$$\mathbf{H}_{n+1}^{-1} = (I - \rho_n \mathbf{y}_n \mathbf{s}_n^T) \mathbf{H}_n^{-1} (I - \rho_n \mathbf{s}_n \mathbf{y}_n^T) + \rho_n \mathbf{s}_n \mathbf{s}_n^T$$

where  $\rho_n = (\mathbf{y}_n^T \mathbf{s}_n)^{-1}$ . Proving this is relatively involved and mostly symbol crunching. I don't know of any intuitive way to derive this unfortunately.

## Broyden, Fletcher, Goldfarb, Shanno





This update is known as the Broyden–Fletcher–Goldfarb–Shanno (BFGS) update, named after the original authors. Some things worth noting about this update:

- $\mathbf{H}_{n+1}^{-1}$  is positive definite (psd) when  $\mathbf{H}_n^{-1}$  is. Assuming our initial guess of  $\mathbf{H}_0$  is psd, it follows by induction each inverse Hessian estimate is as well. Since we can choose any  $\mathbf{H}_0^{-1}$  we want, including the  $I$  matrix, this is easy to ensure.
- The above also specifies a recurrence relationship between  $\mathbf{H}_{n+1}^{-1}$  and  $\mathbf{H}_n^{-1}$ . We only need the history of  $s_n$  and  $y_n$  to re-construct  $\mathbf{H}_n^{-1}$ .

The last point is significant since it will yield a procedural algorithm for computing  $\mathbf{H}_n^{-1}d$ , for a direction  $d$ , without ever forming the  $\mathbf{H}_n^{-1}$  matrix. Repeatedly applying the recurrence above we have

```

BFGSMultiply( $\mathbf{H}_0^{-1}, \{s_k\}, \{y_k\}, d$ ) :
   $r \leftarrow d$ 
  // Compute right product
  for  $i = n, \dots, 1$  :
     $\alpha_i \leftarrow \rho_i s_i^T r$ 
     $r \leftarrow r - \alpha_i y_i$ 
  // Compute center
   $r \leftarrow \mathbf{H}_0^{-1} r$ 
  // Compute left product
  for  $i = 1, \dots, n$  :
     $\beta \leftarrow \rho_i y_i^T r$ 
     $r \leftarrow r + (\alpha_{n-i+1} - \beta) s_i$ 
  return  $r$ 

```

Since the only use for  $\mathbf{H}_n^{-1}$  is via the product  $\mathbf{H}_n^{-1} \mathbf{g}_n$ , we only need the above procedure to use the BFGS approximation in QuasiNewton.



## L-BFGS: BFGS on a memory budget

The BFGS quasi-newton approximation has the benefit of not requiring us to be able to analytically compute the Hessian of a function. However, we still must maintain a history of the  $s_n$  and  $y_n$  vectors for each iteration. Since one of the core-concerns of the **NewtonRaphson** algorithm were the memory requirements associated with maintaining an Hessian, the BFGS Quasi-Newton algorithm doesn't address that since our memory use can grow without bound.





The L-BFGS algorithm, named for *limited* BFGS, simply truncates the **BFGSMultiply** update to use the last  $m$  input differences and gradient differences. This means, we only need to store  $s_n, s_{n-1}, \dots, s_{n-m+1}$  and  $y_n, y_{n-1}, \dots, y_{n-m+1}$  to compute the update. The center product can still use any symmetric psd matrix  $\mathbf{H}_0^{-1}$ , which can also depend on any  $\{s_k\}$  or  $\{y_k\}$ .

## L-BFGS variants

There are lots of variants of L-BFGS which get used in practice. For non-differentiable functions, there is an othant-wise variant (<http://research.microsoft.com/en-us/people/jfgao/paper/icml07scalable.pdf>) which is suitable for training  $L_1$  regularized loss.

One of the main reasons to *not* use L-BFGS is in very large data-settings where an online approach can converge faster. There are in fact online variants (<http://jmlr.org/proceedings/papers/v2/schraudolph07a/schraudolph07a.pdf>) of L-BFGS, but to my knowledge, none have consistently out-performed SGD variants (including AdaGrad (<http://www.magicbroom.info/Papers/DuchiHaSi10.pdf>) or AdaDelta) for sufficiently large data sets.

- 
1. This assumes there is a unique global minimizer for  $f$ . In practice, in practice unless  $f$  is convex, the parameters used are whatever pops out the other side of an iterative algorithm.  $\Leftarrow$
  2. We know  $-\mathbf{H}^{-1}\nabla f$  is a local extrema since the gradient is zero, since the Hessian has positive curvature, we know it's in fact a local minima. If  $f$  is convex, we know the Hessian is always positive definite and we know there is a single unique global minimum.  $\Leftarrow$
  3. As we'll see, we really on require being able to multiply by  $\mathbf{H}^{-1}d$  for a direction  $d$ .  $\Leftarrow$
  4. I've intentionally left the weighting matrix  $W$  used to weight the norm since you get the same solution under many choices. In particular for any positive-definite  $W$  such that  $Ws_n = y_n$ , we get the same solution.  $\Leftarrow$

 (<http://twitter.com/aria42>)  (<http://github.com/aria42>)   
(<http://linkedin.com/in/aria42>)  (<mailto:me@aria42.com>)

© 2016 [aria42.com](http://aria42.com/) (<http://aria42.com/>)