The Levenberg-Marquardt Algorithm

Ananth Ranganathan

8th June 2004

1 Introduction

The Levenberg-Marquardt (LM) algorithm is the most widely used optimization algorithm. It outperforms simple gradient descent and other conjugate gradient methods in a wide variety of problems. This document aims to provide an intuitive explanation for this algorithm. The LM algorithm is first shown to be a blend of vanilla gradient descent and Gauss-Newton iteration. Subsequently, another perspective on the algorithm is provided by considering it as a trust-region method.

2 The Problem

The problem for which the LM algorithm provides a solution is called *Nonlinear Least Squares Minimization*. This implies that the function to be minimized is of the following special form :

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x)$$

where $x = (x_1, x_2, \ldots, x_n)$ is a vector, and each r_j is a function from $\Re^n$ to $\Re$. The r_j are referred to as a *residuals* and it is assumed that $m \geq n$.

To make matters easier, f is represented as a *residual vector* $r : \Re^n \rightarrow \Re^m$ defined by

$$r(x) = (r_1(x), r_2(x), \cdots, r_m(x))$$

Now, f can be rewritten as $f(x) = \frac{1}{2} \parallel r(x) \parallel^2$. The derivatives of f can be written using the Jacobian matrix J of r w.r.t x defined as $J(x) = \frac{\partial r_j}{\partial x_i}$, $1 \leq j \leq m$, $1 \leq i \leq n$.

Let us first consider the linear case where every r_i function is linear. Here, the Jacobian is constant and we can represent r as a hyperplane through space, so that f is given by the quadratic $f(x) = \frac{1}{2}\|Jx + r(0)\|^2$. We also get $\nabla f(x) = J^T(Jx + r)$ and $\nabla^2 f(x) = J^T J$. Solving for the minimum by setting $\nabla f(x) = 0$, we obtain $x_{min} = -(J^T J)^{-1} J^T r$, which is the solution to the set of *normal equations*.

Returning to the general, non-linear case, we have

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \nabla r_j(x) = J(x)^T r(x) \quad (1)$$

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \quad (2)$$

The distinctive property of least-squares problems is that given the Jacobian matrix J, we can essentially get the Hessian ($\nabla^2 f(x)$) for free if it is possible to approximate the r_js by linear functions ($\nabla^2 r_j(x)$ are small) or the residuals ($r_j(x)$) themselves are small. The Hessian in this case simply becomes

$$\nabla^2 f(x) = J(x)^T J(x) \quad (3)$$

which is the same as for the linear case.

The common approximation used here is one of near-linearity of the r_is near the solution so that $\nabla^2 r_j(x)$ are small. It is also important to note that (3) is only valid if the residuals are small. Large residual problems cannot be solved using the quadratic approximation, and consequently, the performance of the algorithms presented in this document is poor in such cases.

3 LM as a blend of Gradient descent and Gauss-Newton iteration

Vanilla gradient descent is the simplest, most intuitive technique to find minima in a function. Parameter updation is performed by adding the negative of the scaled gradient at each step, i.e.

$$x_{i+1} = x_i - \lambda \nabla f \quad (4)$$

Simple gradient descent suffers from various convergence problems. Logically, we would like to take large steps down the gradient at locations where the gradient is small (the slope is gentle) and conversely, take small steps when the gradient is large, so as not to rattle out of the minima. With the above update rule, we do just the opposite of this. Another issue is that the curvature of the error surface may not be the same in all directions. For example, if there is a long and narrow valley in the error surface, the component of the gradient in the direction that points along the base of the valley is very small while the component along the valley walls is quite large. This results in motion more in the direction of the walls even though we have to move a long distance along the base and a small distance along the walls.

This situation can be improved upon by using curvature as well as gradient information, namely second derivatives. One way to do this is to use Newton's method to solve the equation $\nabla f(x) = 0$. Expanding the gradient of f using a Taylor series around the current state x_0, we get

$$\nabla f(x) = \nabla f(x_0) + (x - x_0)^T \nabla^2 f(x_0) + \text{higher order terms of } (x - x_0) \quad (5)$$

If we neglect the higher order terms (assuming f to be quadratic around x_0), and solve for the minimum x by setting the left hand side of (5) to 0, we get the update rule for Newton's method -

$$x_{i+1} = x_i - \left(\nabla^2 f(x_i)\right)^{-1} \nabla f(x_i) \quad (6)$$

where $x_0$has been replaced by x_i and x by x_{i+1}.

Since Newton's method implicitly uses a quadratic assumption on f (arising from the neglect of higher order terms in a Taylor series expansion of f) , the Hessian need not be evaluated exactly. Rather the approximation of (3) can be used. The main advantage of this technique is rapid convergence. However, the rate of convergence is sensitive to the starting location (or more precisely, the linearity around the starting location).

It can be seen that simple gradient descent and Gauss-Newton iteration are complementary in the advantages they provide. Levenberg proposed an algorithm based on this observation, whose update rule is a blend of the above mentioned algorithms and is given as

$$x_{i+1} = x_i - (H + \lambda I)^{-1} \nabla f(x_i) \tag{7}$$

where H is the Hessian matrix evaluated at x_i. This update rule is used as follows. If the error goes down following an update, it implies that our quadratic assumption on $f(x)$ is working and we reduce λ (usually by a factor of 10) to reduce the influence of gradient descent. On the other hand, if the error goes up, we would like to follow the gradient more and so λ is increased by the same factor. The Levenberg algorithm is thus -

1. Do an update as directed by the rule above.
2. Evaluate the error at the new parameter vector.
3. If the error has increased as a result the update, then retract the step (i.e. reset the weights to their previous values) and increase λ by a factor of 10 or some such significant factor. Then go to (1) and try an update again.
4. If the error has decreased as a result of the update, then accept the step (i.e. keep the weights at their new values) and decrease λ by a factor of 10 or so.

The above algorithm has the disadvantage that if the value of λ is large, the calculated Hessian matrix is not used at all. We can derive some advantage out of the second derivative even in such cases by scaling each component of the gradient according to the curvature. This should result in *larger* movement along the directions where the gradient is *smaller* so that the classic "error valley" problem does not occur any more. This crucial insight was provided by Marquardt. He replaced the identity matrix in (7) with the diagonal of the Hessian resulting in the Levenberg-Marquardt update rule.

$$x_{i+1} = x_i - (H + \lambda diag[H])^{-1} \nabla f(x_i) \tag{8}$$

Since the Hessian is proportional to the curvature of f, (8) implies a large step in the direction with low curvature (i.e., an almost flat terrain) and a small step in the direction with high curvature (i.e, a steep incline).

It is to be noted that while the LM method is in no way optimal but is just a heuristic, it works extremely well in practice. The only flaw is its need for matrix inversion as part of the update. Even though the inverse is usually implemented using clever pseudo-inverse methods such as singular value decomposition, the cost of the update becomes prohibitive after the model size increases to a few thousand parameters. For moderately sized models (of a few hundred parameters) however, this method is *much* faster than say, vanilla gradient descent.

4 LM as a trust-region algorithm

Historically, the LM algorithm was presented by Marquardt as given in the previous section where the parameter, λ, was manipulated directly to find the minimum. Subsequently, a trust-region approach to the algorithm has gained ground.

Trust-region algorithms work in a fundamentally different manner than those presented in the previous section, which are called *line-search* methods. In a line search method, we decide on a direction in which to descend the gradient and are then concerned about the step size, i.e. if $p^{(k)}$ is the direction of descent, and α_k the stepsize, then our step is given by $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$ and the stepsize is obtained by solving the sub-problem

$$\min f\left(x^{(k)} + \alpha_k p^{(k)}\right) \forall \alpha_k > 0$$

By contrast, in a trust-region algorithm we build a model $m^{(k)}$ that approximates the function f in a finite region near $x^{(k)}$. This region, Δ, where the model is a good approximation of f, is called the trust-region. Trust-region algorithms maintain Δ and update it at each iteration using heuristics. The model $m^{(k)}$ is most often a quadratic obtained by a Taylor series expansion of f around $x^{(k)}$, i.e.

$$m^{(k)} = f\left(x^{(k)}\right) + \nabla f\left(x^{(k)}\right) \cdot p + \frac{1}{2} p^T H p \tag{9}$$

where H is the Hessian (or an approximation of the Hessian) matrix. The sub-problem to be solved to find the step to take during the iteration is

$$min_{\|p\| \leq \Delta} f\left(x^{(k)}\right) + \nabla f\left(x^{(k)}\right) \cdot p + \frac{1}{2} p^T H p \tag{10}$$

and the iteration step itself is $x^{(k+1)} = x^{(k)} + p$. A trust-region algorithm can thus be conceived of as a sequence of iterations, in each of which we model the function f by a quadratic and then jump to the minimum of that quadratic.

The solution of (10) is given by a theorem which is as follows -

> p^* is a global solution of $min_{\|p\| < \Delta} f\left(x^{(k)}\right) + \nabla f\left(x^{(k)}\right) \cdot p + \frac{1}{2} p^T H p$ iff
>
> $\| p^* \| \leq \Delta$ and there is a scalar λ s.t. :
>
> $$(H + \lambda I)\, p^* = -g \tag{11}$$
> $$\lambda\, (\Delta - p^*) = 0 \tag{12}$$
>
> and $(H + \lambda I)$ is positive semi-definite
>
> where $g, f \in \Re^n$.

It can be seen that (11) is the same as (7). (12) basically states that if $\| p^* \| < \Delta$ then λ is 0 but not otherwise. Hence, we reach the same parameter update equation for the LM algorithm using a trust-region framework as we obtained using the line-search method

The heuristic to update the size of the trust-region usually depends on the ratio of the expected change in f to the predicted change, i.e.

$$\rho_k = \frac{f\left(w^{(k)}\right) - f\left(w^{(k)} + p^*\right)}{f\left(w^{(k)}\right) - m^{(k)}\left(p^*\right)} \tag{13}$$

If there is a good agreement between predicted and actual values ($\rho_k \approx 1$), then Δ is increased; if the agreement is poor (ρ_k is small), then ρ_k is decreased. If ρ_k is smaller than a threshold value ($\sim 10^{-4}$), the step is rejected and the value of w_k is retained but Δ is decreased as before. Thus, the algorithm is similar to the one in Section 3 but the value that is changed with each iteration is Δ and not λ.

References

[Levenberg44] K. Levenberg, "A method for the solution of certain problems in least squares, *Quart. Appl. Math.*, 1944, Vol. 2, pp. 164–168.

[Marquardt63] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math.*, 1963, Vol. 11, pp. 431–441.

[Nocedal99] J. Nocedal and S.J. Wright, "Numerical Optimization," *Springer*, New York, 1999.