Levenberg-Marquardt Optimization

Sam Roweis

Abstract

Levenberg-Marquardt Optimization is a virtual standard in nonlinear optimization which significantly outperforms gradient descent and conjugate gradient methods for medium sized problems. It is a pseudo-second order method which means that it works with only function evaluations and gradient information but it estimates the Hessian matrix using the sum of outer products of the gradients.

This note reviews the mathematical motivations for Levenberg-Marquardt and also details the algorithm.

0.1 Improvements to simple gradient descent

Vanilla gradient descent works fine for very simple models, but is too simplistic a method for more complex models may free parameters. Convergence can take an extremely long time: the key insight into why is to notice that the problem is *stiff* in the sense that the few places where small step sizes are required ruin it for the whole problem. For example, when descending the walls of a very steep local minimum bowl we must use a very small step size to avoid "rattling out" of the bowl. On the other hand when we are moving along a gently sloping part of the error surface we want to take large steps otherwise it will take forever to get anywhere. This problem is compounded by the strange manner in which we implement gradient descent – we normally move by making a step that is some constant times the negative gradient rather than a step of constant length in the direction of the negative gradient. This means that in steep regions (where we have to be careful not to make our steps too large) we move quickly and in shallow regions (where we need to move in big steps) we move slowly. 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 base of the valley is very small while the component perpendicular to the valley walls is quite large even though we have to move a long distance along the base and a small distance perpendicular to the walls. So we would like to use slightly more sophisticated gradient descent algorithms than simple steepest descent, which is just

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \mu \nabla E(\mathbf{w})$$

This can be greatly improved upon with a little cleverness. Using *second order* information – in other words using the curvature as well as the gradient of the error surface – can speed things up enormously. However, it is often prohibitively expensive to compute the second derivatives of a model exactly. Many successful techniques rely on *estimating* some curvature information from only function evaluations and first order derivatives (gradients). For example, adding *momentum* to the

weight changes alleviates many of the above problems. Momentum is an example of an improvement on our simple first order method that keeps it first order but tries to get some curvature information by averaging the gradients locally. However in order to cut down on cost it only averages the gradients that have already been computed. Another technique which improves basic gradient descent by estimating curvature information and which works extremely well for medium sized models is discussed below.

0.2 The quadratic approximation of E

We saw that when we have a linear function model, then the error function E was a simple quadratic form. Recall that we define E as the average squared error:

$$E(\mathbf{w}) =< (f(\mathbf{x};\mathbf{w}) - y)^2 >$$

where the angle brackets donate the mean over input output pairs. For linear functions, this had a quadratic form:

$$E(\mathbf{w}) = a + 2\mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$$

where a, $\mathbf{b}$, and $\mathbf{C}$ depend on averages over the input output pairs. We can find the minimum in closed form by solving for when the gradient goes to zero. Don't be concerned by all the matrix notation, this is exactly the same thing that you know well from the scalar case: if the function is linear then squared error is quadratic and everybody knows how to solve for the minimum of a parabola.

$$\nabla E(\mathbf{w}) = \mathbf{b} + \mathbf{C}\mathbf{w}$$

$$\nabla E(\mathbf{w}) = 0 \Rightarrow \mathbf{b} + \mathbf{C}\mathbf{w}_{opt} = 0$$

$$-\mathbf{b} = \mathbf{C}\mathbf{w}_{opt}$$

$$\mathbf{w}_{opt} = -\mathbf{C}^{-1}\mathbf{b}$$

Here we don't have to do any gradient descent, we can just jump directly to the minimum. For nonlinear choices of f of course, our function E will be more complex than the above quadratic form and we won't just be able to hop to a minimum. However, close to a minimum, we can *linearize* the function which will approximate E by a quadratic equation and use the above method to guess where the minimum is. Even if we don't jump right to the true minimum the first time, if the approximation is good we will converge much more quickly than with steepest descent. In essence what we are doing is looking at the curvature of the error surface where we are and assuming that the curvature we see is due to a parabolic bowl. Then we jump to the bottom of this fictitious bowl and re-evaluate things wherever we end up.

We will derive this quadratic approximation to E in a series of steps. First, consider the general deterministic model $f(\mathbf{x}; \mathbf{w})$. It is a function of both the data $\mathbf{x}$ and its parameters $\mathbf{w}$. When we *use* our model to predict the behavior of the unknown target function, we have fixed $\mathbf{w}$ and are more interested in f as a function of $\mathbf{x}$. However, when we are *training* our model by optimizing the weights to reduce E, we are interested in f as a function of $\mathbf{w}$. For the discussion below, we will concentrate on this view of f, and all derivatives and gradients will be with respect to $\mathbf{w}$.

One way to approximate E as locally quadratic (in $\mathbf{w}$) near a minimum is to approximate $f(\mathbf{x}; \mathbf{w})$ as a linear function of $\mathbf{w}$, which we will now derive. Remember that all gradients are with respect to $\mathbf{w}$ and all averages are over input output pairs.

• First we write a new function $\hat{f}(\mathbf{x}; \mathbf{w})$ which is a linear approximation of $f(\mathbf{x}; \mathbf{w})$ in the neighborhood of a specific weight value $\mathbf{w}_0$:

$$\hat{f}(\mathbf{x}; \mathbf{w}) = f(\mathbf{x}; \mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \nabla f(\mathbf{x}; \mathbf{w}_0)$$

• Assuming the model is $\hat{f}$, we write expressions for $E(\mathbf{w})$ and $\nabla E(\mathbf{w})$ in terms of y, $\hat{f}(\mathbf{x}; \mathbf{w})$ and $\nabla \hat{f}(\mathbf{x}; \mathbf{w})$:

$$\hat{E}(\mathbf{w}) = \left\langle (\hat{f}(\mathbf{x}; \mathbf{w}) - y)^2 \right\rangle$$

$$\nabla \hat{E}(\mathbf{w}) = \left\langle 2(\hat{f}(\mathbf{x}; \mathbf{w}) - y) \nabla \hat{f}(\mathbf{x}; \mathbf{w}) \right\rangle$$

• Substituting for $\hat{f}$, we solve for $\nabla \hat{E}(\mathbf{w})$ in terms of y, $\mathbf{w}$, $\mathbf{w}_0$, $f(\mathbf{x}; \mathbf{w}_0)$, and $\nabla f(\mathbf{x}; \mathbf{w}_0)$. (First note the simple form of $\nabla \hat{f}(\mathbf{x}; \mathbf{w})$: $\nabla \hat{f}(\mathbf{x}; \mathbf{w}) = \nabla f(\mathbf{x}; \mathbf{w}_0)$ because $\hat{f}$ is a linear function of $\mathbf{w}$.) Now we can solve for $\nabla \hat{E}$,

$$\nabla \hat{E}(\mathbf{w}) = \left\langle 2(f(\mathbf{x}; \mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \nabla f(\mathbf{x}; \mathbf{w}_0) - y) \nabla f(\mathbf{x}; \mathbf{w}_0) \right\rangle$$

Let

$$\mathbf{d} = \langle (f(\mathbf{x}; \mathbf{w}_0) - y) \nabla f(\mathbf{x}; \mathbf{w}_0) \rangle$$

$$\mathbf{H} = \left\langle \nabla f(\mathbf{x}; \mathbf{w}_0) \nabla f(\mathbf{x}; \mathbf{w}_0)^T \right\rangle$$

where the letter $\mathbf{d}$ stands for *derivative* and the letter $\mathbf{H}$ stands for *Hessian*. Notice that while $\mathbf{d}$ is exactly the average error gradient, $\mathbf{H}$ is not the true Hessian (matrix of mixed partials) of the function. In other words $H_{ij} \neq \partial f / \partial x_i \partial x_j$. Instead, $\mathbf{H}$ is an *approximation* to the Hessian which is obtained by averaging outer products of the first order derivative (gradient). This approximation is exact if f is linear, but in general may be quite poor. However, the trick that we will soon see is that we rely on this approximation only in regions where a linear approximation to f is reasonable. One final point: it may look as though $\mathbf{H}$ is of rank one since it is made from outer products of

vectors, however remember that it is the *average* of many such outer products (each of rank one) and so in general it is full rank.

Back to our derivation to finish up the quadratic approximation:

• We write the previous equation in terms of $\mathbf{d}$ and $\mathbf{H}$, then solve for the $\mathbf{w}$ where $\nabla \hat{E}$ goes to zero.

$$\nabla \hat{E}(\mathbf{w}) = 2\mathbf{H}(\mathbf{w} - \mathbf{w}_0) + 2\mathbf{d}$$

$$\nabla \hat{E}(\mathbf{w}) = 0 \Rightarrow 2\mathbf{H}(\mathbf{w}_{opt} - \mathbf{w}_0) + 2\mathbf{d} = 0$$

$$\mathbf{w}_{opt} = -\mathbf{H}^{-1}\mathbf{d} + \mathbf{w}_0$$

0.3 The Levenberg gradient descent method

Now that we are finished with the derivations, it is time to put all of this back into perspective. We began this note with the idea that we could improve upon steepest descent. Given our definition of $\mathbf{d}$, steepest descent is simply

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \mu\mathbf{d}$$

Compare this to the update rule based on our quadratic approximation

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \mathbf{H}^{-1}\mathbf{d}$$

Our quadratic rule is not universally better since it assumes a linear approximation of f's dependence on $\mathbf{w}$, which is only valid near a minimum. The technique invented by Levenberg involves "blending" between these two extremes. We can use a steepest descent type method until we approach a minimum, then gradually switch to the quadratic rule. We can try to guess how close we are to a minimum by how our error is changing. In particular, Levenberg's algorithm is formalized as follows: let λ be a blending factor which will determine the mix between steepest descent and the quadratic approximation. The update rule is

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{d}$$

where $\mathbf{I}$ is the identity matrix. As λ gets small, the rule approaches the quadratic approximation update rule above. If λ is large, the rule approaches

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \frac{1}{\lambda}\mathbf{d}$$

which is steepest descent. The algorithm adjusts λ according to whether E is increasing or decreasing as follows:

1. Do an update as directed by the rule above.
2. Evaluate the error at the new weight 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 intuition is that if error is increasing, our quadratic approximation is not working well and we are likely not near a minimum, so we should increase λ in order to blend more towards simple gradient descent. Conversely, if error is decreasing, our approximation is working well, and we expect that we are getting closer to a minimum so λ is decreased to bank more on the Hessian.

Marquardt improved this method with a clever incorporation of estimated local curvature information, resulting in the *Levenberg-Marquardt* method. The insight of Marquardt was that when λ is high and we are doing essentially gradient descent, we can still get some benefit from the Hessian matrix that we estimated. In essence, he suggested that we should move *further* in the directions in which the gradient is *smaller* in order to get around the classic "error valley" problem. So he replaced the identity matrix in Levenberg's original equations with the diagonal of the Hessian:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\mathbf{H} + \lambda diag[\mathbf{H}])^{-1}\mathbf{d}$$

As you can see, all this method needs is to operate are the same things steepest descent needs: y, $f(\mathbf{x}, \mathbf{w})$ and $\nabla f(\mathbf{x}; \mathbf{w})$. In other words, we can compute $\mathbf{d}$ and $\mathbf{H}$ based only on the value of the function and its gradient which we know how to evaluate. Don't confuse this with being an alternative to backprop: it is an alternative to simple gradient descent. Backprop is nothing more than a clever and efficient algorithm for evaluating $\nabla f(\mathbf{x}; \mathbf{w})$ for networks. We can then use this gradient in any way we want, either to do steepest descent or to do something tricky like Levenberg-Marquardt.

It is also important to know that this is nothing more than a heuristic method. It is not optimal for any well defined criterion of speed or final error, it is merely a well thought out optimization procedure. But *it is one that works extremely well in practice.* It has become a virtual standard for optimization of medium sized nonlinear models. Its only flaw is that it requires a matrix inversion step as part of the update which scales as the N^3 where N is the number of weights. For medium sized networks (a few hundred weights say) this method will be *much much* faster than gradient descent plus momentum. However for a few thousand weights the cost of matrix inversion begins to kill us and the speed gained by the cleverness of the method is lost in the time taken to do each iteration. It is also important to be aware that the innocuous looking inverse operation is tricky to implement in practice: it can be ill-conditioned and pseudo-inverse methods such as the singular value decomposition are almost always preferred over a naive vanilla inverse. The gory details of how this works are reviewed in more detail but somewhat less clarity in *Numerical Recipes in C, 2nd Edition*, pages 683-685.