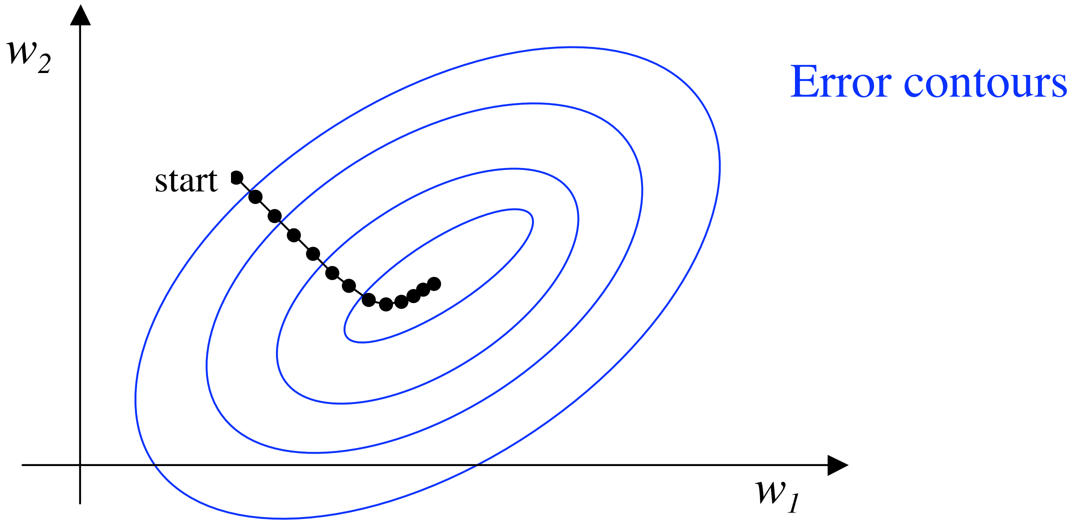
Learning with Momentum, Conjugate Gradient Learning

Neural Computation : Lecture 8

Visualising Learning

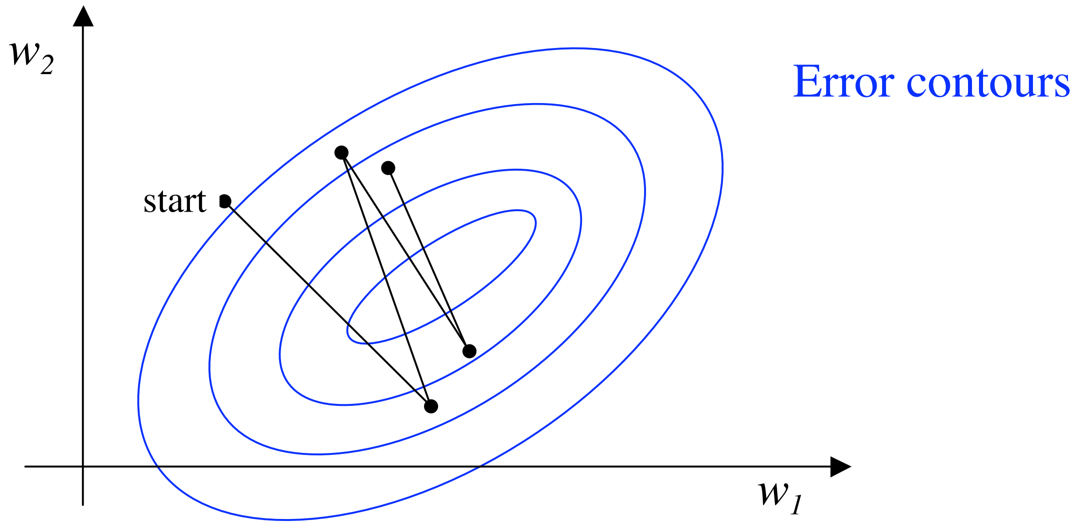
Visualising neural network learning is difficult because there are so many weights being updated at once. However, one can plot error function contours for pairs of weights to get some idea of what is happening. The weight update equations will produce a series of steps in weight space from the starting position to an error minimum:



True gradient descent produces a smooth curve perpendicular to the contours. Weight updates with a small step size η will result in a good approximation to it as shown.

Step Size Too Large

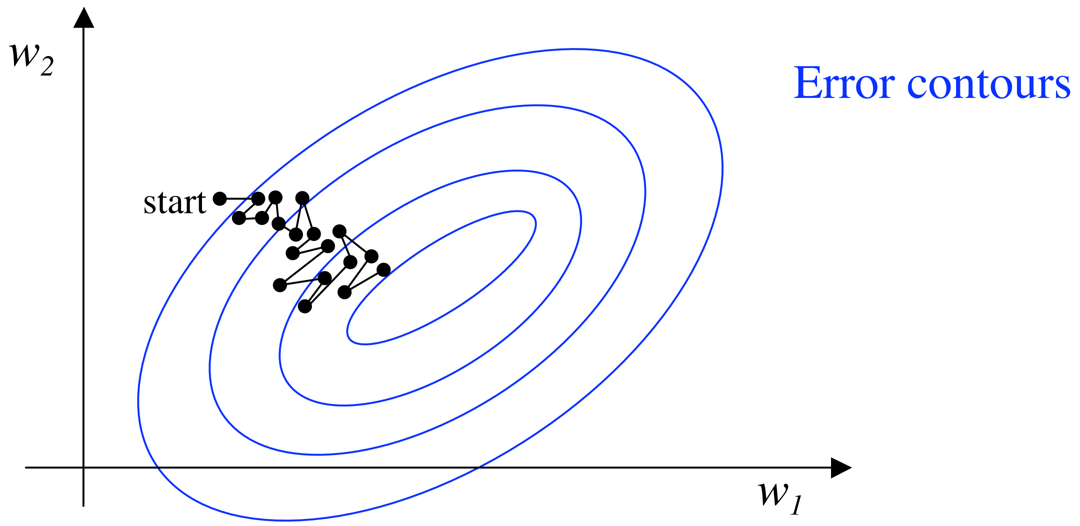
If the step size (i.e. learning rate η) is set too large, the approximation to true gradient descent will be poor, and this will result in overshoots, or even divergence:



In practice, there is no need to plot error contours to see if this is happening. It is obvious that, if the error function is fluctuating, it would be wise to reduce the step size. It is also worth checking individual weights and output activations as well. On the other hand, if everything is smooth, it is worth trying to increase η and seeing if it stays smooth.

On-line Learning

If the weights are updated after each training pattern, rather than adding up the weight changes for all the patterns before applying them, the learning algorithm is no longer true gradient descent, and the weight changes will not be perpendicular to the contours:



If the step sizes are kept small enough, the erratic behaviour of the weight updates will not be too much of a problem, and the increased number of weight changes will still get us to the minimum quicker than true gradient descent (i.e. batch learning).

Learning with Momentum

A compromise that will smooth out the erratic behaviour of on-line updates, without slowing down the learning too much, is to update the weights with the ***moving average*** of the individual weight changes corresponding to single training patterns.

If everything is labelled by the time t (which can conveniently be measured in weight update steps), then implementing a moving average is easy:

$$\Delta w_{hl}^{(n)}(t) = \eta.delta_l^{(n)}(t).out_h^{(n-1)}(t) + \alpha.\Delta w_{hl}^{(n)}(t-1)$$

One simply adds a ***momentum*** term $\alpha.\Delta w_{hl}^{(n)}(t-1)$ which is the weight change of the previous step times a momentum parameter α. If α is zero, then we have the standard on-line training algorithm used before. As α is increased towards one, each step includes increasing contributions from the previous training patterns. Obviously it makes no sense to have α less than zero, or greater than one. Good sizes of α depend on the size of the training data set and how variable it is. Usually, we will need to decrease η as we increase α, so that the total step sizes don't get too large.

Learning with Line Searches

Learning algorithms which work by taking a sequence of steps through weight space all have two basic components: the ***step size*** $size(t)$ and the ***direction*** $dir_{hl}^{(n)}(t)$ such that

$$\Delta w_{hl}^{(n)}(t) = size(t).dir_{hl}^{(n)}(t)$$

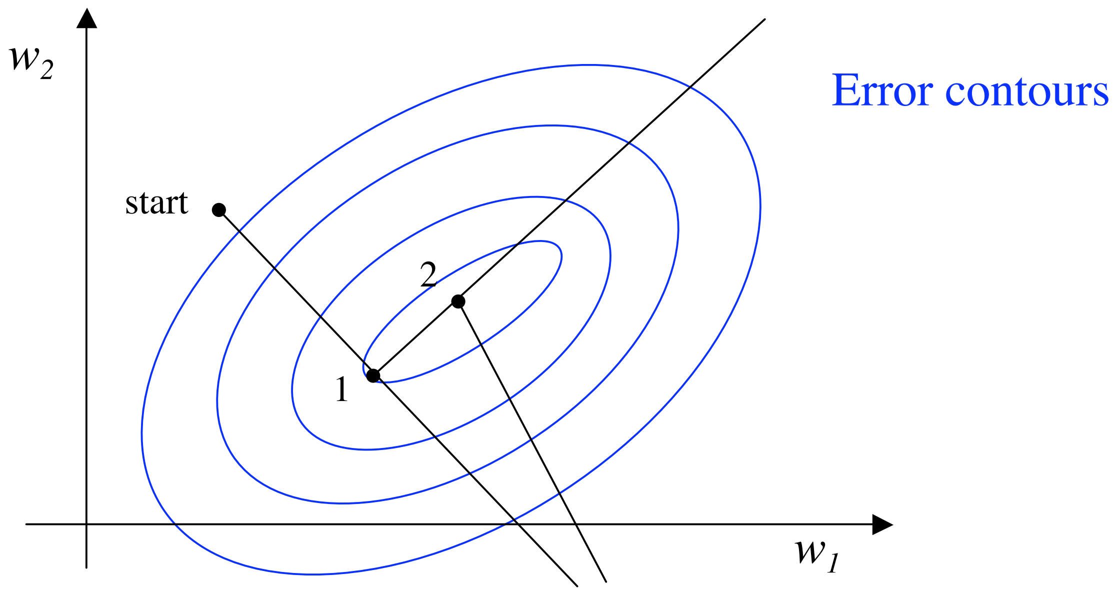
For gradient descent algorithms, such as Back-Propagation, the chosen direction is given by the partial derivatives of the error function $dir_{hl}^{(n)}(t) = -\,\partial E(w_{jk}^{(m)}) \big/ \partial w_{hl}^{(n)}$, and the step size is simply the small constant learning rate parameter $size(t)=\eta$.

A better procedure might be to carry on along in the chosen direction until the error starts rising again. This involves performing a ***line search*** to determine the step size.

The simplest procedure for performing a line search would be to take a series of small steps along the chosen direction until the error increases, and then go back one step. However, that is not likely to be any more efficient than standard gradient descent. Fortunately, there exist better procedures for performing line searches.

Determining the Step Size

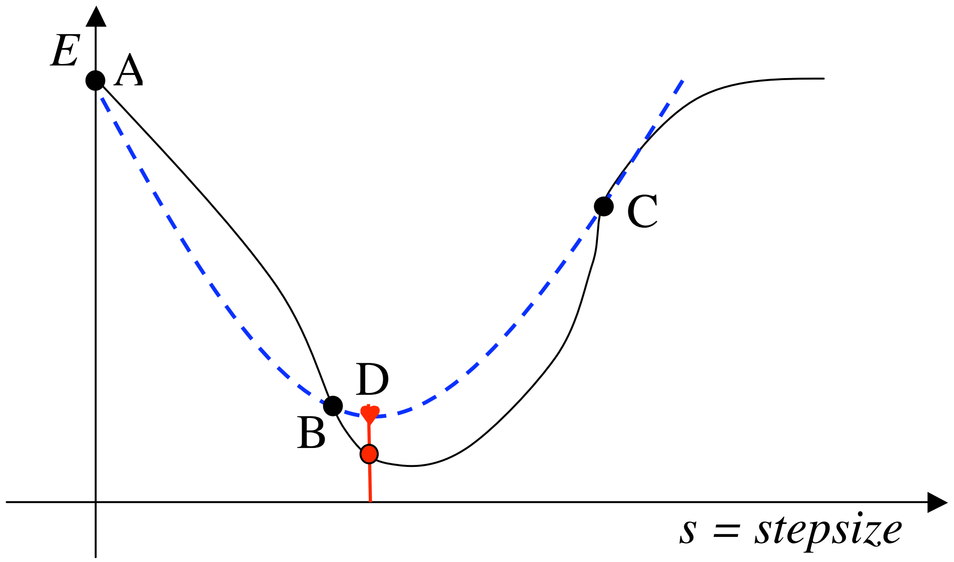
It is possible to see how to proceed with choosing an optimal step size by looking again at the error contours plot of a typical learning process:



At each stage (start, 1, 2, ...) the gradient line defines a one dimensional section through the error surface. There are well known iterative procedures for finding the minimum of such a section – a particularly good one is called ***Parabolic Interpolation***.

Parabolic Interpolation

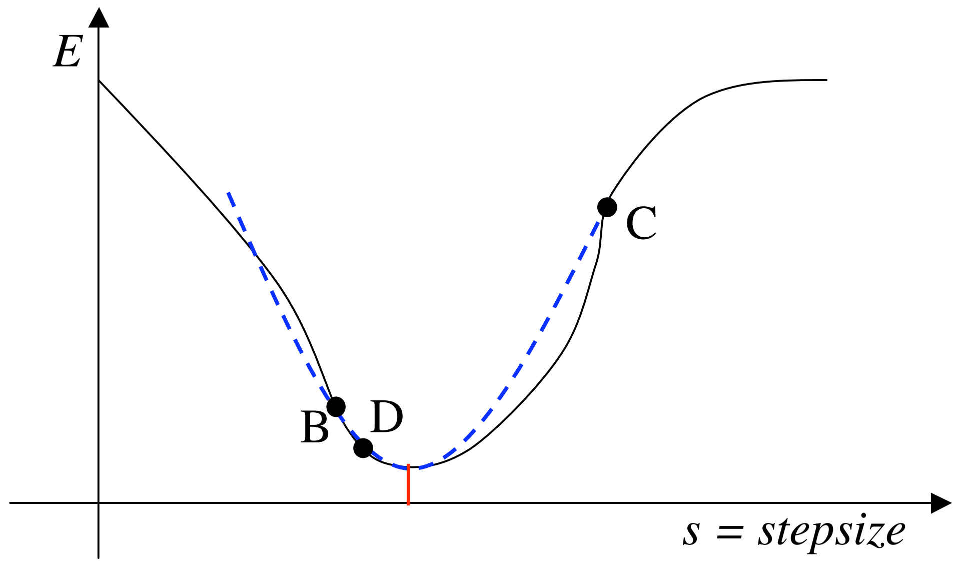
One can plot the variation of the error function *E(s)* as the step *s* taken along the chosen direction is increased. Suppose we can find three points A, B, C such that $E(\mathrm{A}) > E(\mathrm{B})$ and $E(\mathrm{B}) < E(\mathrm{C})$. If A is where we are, and C is far away, this should not be difficult.



The three points are sufficient to define the parabola that passes through them, and we can then easily compute the minimum point D of that parabola. The step size corresponding to that point D is a good guess for the appropriate step size for the actual error function.

Repeated Parabolic Interpolation

The process of parabolic interpolation can easily be repeated. We take the guess of the appropriate step size given by point D, together with the two original points of lowest error (i.e. B and C) to make up a new set of three points, and repeat the procedure:



Each iteration of this process brings us closer to the minimum. However, the gradients change with each step, so it is not computationally efficient to find each step size too accurately. Usually it is better to get it roughly right and move on to the next direction.

Properties of the Gradient Descent Directions

We have seen how we can determine appropriate step sizes by doing line searches, but it is not obvious that using the gradient descent direction is really the best thing to do.

Since the step size $s(t\!-\!1)$ is chosen to minimise the new error $E(w_{ij}(t))$, the gradient of that error with respect to that step size must be zero at the new weight $w_{ij}(t)$, i.e.

$$\frac{\partial E(w_{ij}(t))}{\partial s(t-1)} = 0$$

Also, using the standard chain rule for derivatives, we know:

$$\frac{\partial E(w_{ij}(t))}{\partial s(t-1)} = \sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial w_{ij}(t)}{\partial s(t-1)}$$

so the gradient descent directions $-\partial E(w_{ij}(t)) \big/ \partial w_{ij}(t)$ are seen to have the property:

$$\sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial w_{ij}(t)}{\partial s(t-1)} = 0$$

Problems using Gradient Descent with Line Search

To see what that implies, we need to remember that the new weights $w_{ij}(t)$ are

$$w_{ij}(t) = w_{ij}(t-1) - s(t-1)\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}$$

and hence the partial derivatives of those with respect to the step size $s(t-1)$ are

$$\frac{\partial w_{ij}(t)}{\partial s(t-1)} = -\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}$$

Substituting that into the above gradient descent direction property reveals that:

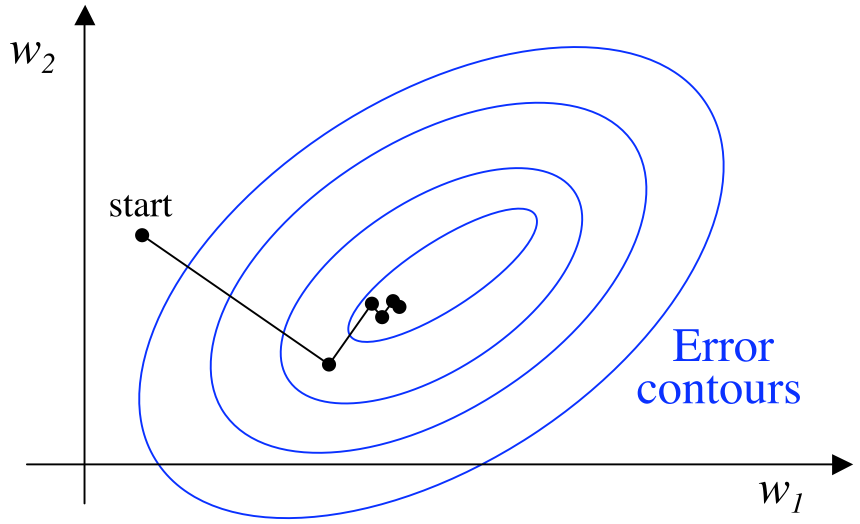
$$\sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} . \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} = 0$$

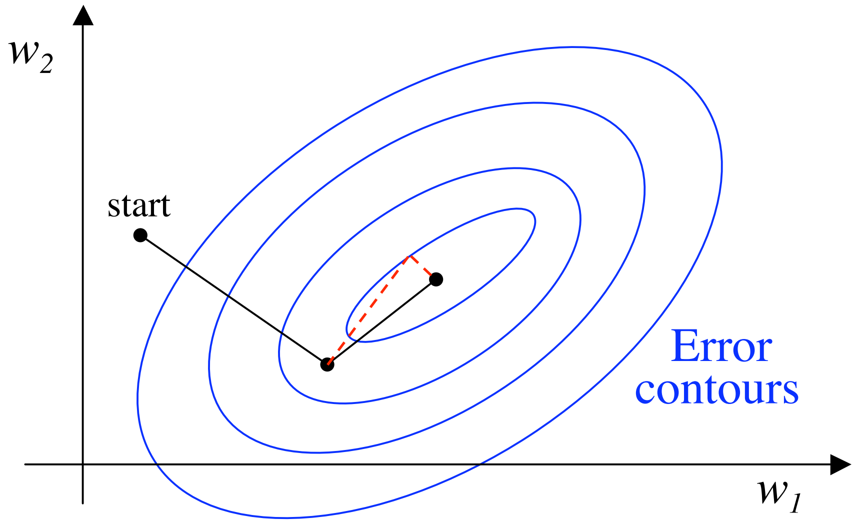
The scalar product of the old direction $-\partial E(w_{ij}(t-1)) \big/ \partial w_{ij}(t-1)$ and the new direction $-\partial E(w_{ij}(t)) \big/ \partial w_{ij}(t)$ is zero, which means the directions are orthogonal (perpendicular). This will result in a less than optimal zig-zag path through weight space.

Finding a Better Search Direction

An obvious approach to avoid the zig-zagging that occurs when we use line searches and gradient directions is to make the new step direction $dir_{ij}(t)$ a compromise between the new gradient direction $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ and the previous step direction $dir_{ij}(t-1)$:

$$dir_{ij}(t) = -\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} + \beta . dir_{ij}(t-1)$$





Conjugate Gradient Learning

The basis of ***Conjugate Gradient Learning*** is to find a value for β in the last equation so that each new search direction spoils as little as possible the minimisation achieved by the previous one. We thus want to find the new direction $dir_{ij}(t)$ such that the gradient $-\partial E(w_{ij}(t))\big/\partial w_{ij}(t)$ at the new point $w_{ij}(t)+s.dir_{ij}(t)$ in the old direction is zero, i.e.

$$\sum_{i,j} dir_{ij}(t-1).\frac{\partial E(w_{ij}(t)+s.dir_{ij}(t))}{\partial w_{ij}(t)}=0$$

The appropriate value of β that satisfies this is given by the ***Polak-Ribiere rule***

$$\beta=\frac{\sum_{i,j}\left(\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}-\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}\right).\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j}\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}.\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}}$$

For most practical applications this is the fastest way to train a neural network.

Approximating the Error Surface

The idea of a quadratic (parabolic) approximation along a line search can be extended to the whole error surface. We can perform a ***Taylor Expansion*** of the cost function around the current point $w_i(t)$ in weight space:

$$E(w_i(t)+\Delta w_i(t)) = E(w_i(t)) + \sum_j \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} \Delta w_j(t)$$

$$+\frac{1}{2}\sum_k \sum_j \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_j(t)\Delta w_k(t) + 0(\Delta w(t)^3)$$

The first derivative is the ***Gradient Vector*** G_j that we have been using before, and the second derivative is known as the local ***Hessian Matrix*** H_{jk}

$$G_j = \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} \quad , \quad H_{jk} = \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)}$$

This approximation can be used as a basis of powerful learning algorithms.

Newton Method for Weight Updating

From the Taylor expansion, the local approximation for the error gradient is

$$\frac{E(w_i(t)+\Delta w_i(t)) - E(w_i(t))}{\Delta w_j(t)} = \left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} + \sum_k \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_k(t) + 0(\Delta w(t)^2)$$

The first term is what we use for the standard gradient descent learning algorithm. If we use the Hessian term too, we can expect a better learning algorithm. If we ignore the higher order terms in *Δw* and note that the error gradient at the minimum is zero, then

$$\left.\frac{\partial E(w_i)}{\partial w_j}\right|_{w_i(t)} + \sum_k \left.\frac{\partial^2 E(w_i)}{\partial w_j \partial w_k}\right|_{w_i(t)} \Delta w_k(t) = 0$$

If the Hessian is invertible, we can solve this to give the ***Newton Method*** weight update

$$\Delta w_k(t) = -\sum_j G_j H_{jk}^{-1}$$

Quasi-Newton Method for Weight Updating

Unfortunately, the Newton Method is usually computationally prohibitive, requiring $0(NW^2)$ operations to compute the Hessian and $0(W^3)$ operations to invert it (where W is the number of weights and N is the number of training patterns).

Quasi-Newton Methods use second order (curvature) information about the error surface without requiring computation of the Hessian itself. They build up an approximation to the inverse Hessian H^{-1} or weight update vector GH^{-1} over a number of steps, using successive iterations of the weights $w(t)$ and gradients $G(t)$.

There are numerous approaches in the literature for doing this. Which approach is best will depend on how well the error surface approximates a quadratic, and how many weights the network has.

Like the Conjugate Gradient method, these methods can find the minimum of a quadratic error surface in at most W steps with an overall computational cost of $0(NW^2)$.

Extreme Learning Machines

The more sophisticated learning algorithms presented above learn more efficiently, but they come at the cost of more complexity, and associated difficulty in programming.

An alternative strategy, that has been reinvented numerous times but is now commonly known as the ***Extreme Learning Machine*** (ELM), simply assigns the input-to-hidden weights at random, and then computes the hidden-to-output weights analytically using the matrix pseudo-inverse approach described previously for single layer networks.

This can work because the random weights are likely to project the input patterns into a linearly separable higher dimensional hidden layer representation, and the output weights are able to find an optimal way of using that representation to produce the outputs.

The main advantage of this approach is that it is very fast compared with gradient descent learning, there are no learning rates to fix, and it will not suffer the problem of local minima. However, it is debatable how good the generalization performance is.

Overview and Reading

1. We began by looking at how we can visualise the process of stepping through weight space to find the error minimum.
2. We saw how adding a momentum term to the gradient descent weight update equations can smooth out and speed up on-line training.
3. Then we looked at using Line Search approaches to training, and the Conjugate Gradient learning algorithm.
4. We ended with an outline of Newton, Quasi-Newton and ELM methods.

Reading

1. Bishop: Sections 4.8, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10
2. Haykin-2009: Sections 4.4, 4.15, 4.16, 4.20
3. Hertz, Krogh & Palmer: Sections 6.1, 6.2
4. Gurney: Sections 5.4, 6.5