Chapter 8

Faster Supervised Learning

8.1 Objectives

After this Chapter, you should

1. understand the innate difficulty with error descent.
2. know several methods of improving convergence.
3. be able to implement a Radial Basis Function network.
4. be able to compare supervised learning methods.

8.2 Radial Basis Functions

We noted in Chapter 4 that, while the multilayer perceptron is capable of approximating any continuous function, it can suffer from excessively long training times. In this chapter we will investigate methods of shortening training times for artificial neural networks using supervised learning.

A typical radial basis function (RBF) network is shown in Figure 8.1. The input layer is simply a receptor for the input data. The crucial feature of the RBF network is the function calculation which is performed in the hidden layer. This function performs a *non-linear* transformation from the input space to the hidden-layer space. The hidden neurons' functions form a basis for the input vectors and the output neurons merely calculate a linear (weighted) combination of the hidden neurons' outputs.

An often-used set of basis functions is the set of Gaussian functions whose mean and standard deviation may be determined in some way by the input data (see below). Therefore, if $\phi(\mathbf{x})$ is the vector of hidden neurons' outputs when the input pattern $\mathbf{x}$ is presented and if there are M hidden neurons, then

$$\begin{aligned} \phi(\mathbf{x}) &= (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), ..., \phi_M(\mathbf{x}))^T \\ \text{where } \phi_i(\mathbf{x}) &= \exp\left(-\lambda_i \parallel \mathbf{x} - \mathbf{c}_i \parallel^2\right) \end{aligned}$$

where the centres $\mathbf{c}_i$ of the Gaussians will be determined by the input data. Note that the terms $\parallel \mathbf{x} - \mathbf{c}_i \parallel$ represent the Euclidean distance between the inputs and the i^{th} centre. For the moment we will only consider basis functions with $\lambda_i = 1$. The output of the network is calculated by

$$y = \mathbf{w}.\phi(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) \tag{8.1}$$

where $\mathbf{w}$ is the weight vector from the hidden neurons to the output neuron.

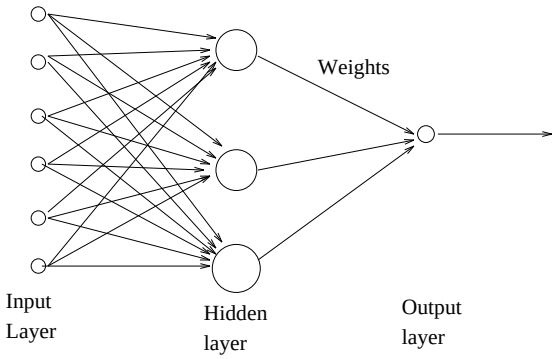

Figure 8.1: A typical radial basis function network. Activation is fed forward from the input layer to the hidden layer where a (basis) function of the Euclidean distance between the inputs and the centres of the basis functions is calculated. The weighted sum of the hidden neuron's activations is calculated at the single output neuron

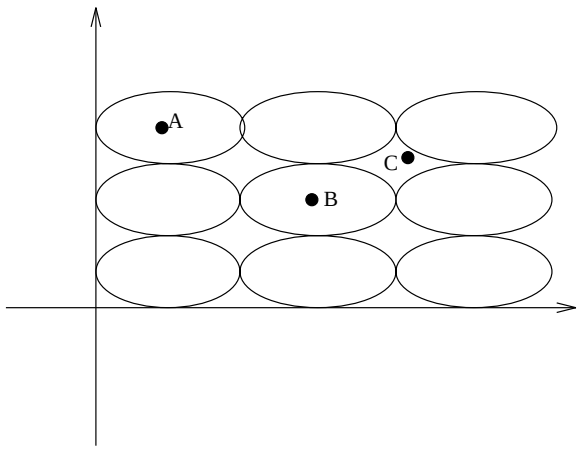

Figure 8.2: A "tiling" of a part of the plane

Input pattern	$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$
(0,0)	0.135	1
(0,1)	0.368	0.368
(1,0)	0.368	0.368
(1,1)	1	0.135

Table 8.1: The activation functions of the hidden neurons for the 4 possible inputs for the XOR problem

To get some idea of the effect of basis functions consider Figure 8.2. In this figure we have used an eliptical tiling of a portion of the plane; this could be thought of as a Gaussian tiling as defined above but with a different standard deviation in the vertical direction from that in the horizontal direction. We may then view the lines drawn as the 1 (or 2 or 3 ...) standard deviation contour. Then each basis function is centred as shown but each has a non-zero effect elsewhere. Thus we may think of
A as the point (1,0,0,0,0,0,0,0,0)
and B as (0,0,0,0,1,0,0,0,0)
Since the basis functions actually have non-zero values everywhere this is an approximation since A will have some effect particularly on the second, fourth and fifth basis functions (the next three closest) but these values will be relatively small compared to 1, the value of the first basis function.

However the value of the basis functions marked 2,3,5 and 6 at the point C will be non-negligible. Thus the coordinates of C in this basis might be thought of as (0,0.2,0.5,0,0.3,0.4,0,0,0) i.e. it is non-zero over 4 dimensions.

Notice also from this simple example that we have increased the dimensionality of each point by using this basis.

We will use the XOR function to demonstrate that expressing the input patterns in the hidden layer's basis permits patterns which were not linearly separable in the original (input) space to become linearly separable in the hidden neurons' space (see Haykin page 241).

8.2.1 XOR Again

We will use the XOR pattern which is shown diagrammatically in Figure 3.4. We noted earlier that this set of patterns is not linearly separable (in the input (X,Y)-space). Let us consider the effect of mapping the inputs to a hidden layer with two neurons with

$$\begin{aligned} \phi_1(\mathbf{x}) &= \exp(-\parallel \mathbf{x}-\mathbf{c}_1 \parallel^2), \text{ where } \mathbf{c}_1 = (1,1) \\ \phi_2(\mathbf{x}) &= \exp(-\parallel \mathbf{x}-\mathbf{c}_2 \parallel^2), \text{ where } \mathbf{c}_2 = (0,0) \end{aligned}$$

Then the two hidden neurons' outputs on presentation of the four input patterns is shown in Table 8.1. Now if we plot the hidden neuron's outputs in the ϕ_1, ϕ_2 basis, we see (Figure 8.3) that the outputs are linearly separable.

8.2.2 Learning weights

However we still have to find the actual parameters which determine the slope of the discrimination line. These are the weights between the basis functions (in the hidden layer) and the output layer.

We may train the network now using the simple LMS algorithm (see Chapter 3) in the usual way. If the sum of the errors over all P patterns is

$$E = \frac{1}{2}\sum_{k=1}^{P}(D^k - y^k)^2 = \frac{1}{2}\sum_{k=1}^{P}(D^k - \sum_j w_j\phi_j(\mathbf{x}^k))^2 = \frac{1}{2}\sum_{k=1}^{P}(e^k)^2 \tag{8.2}$$

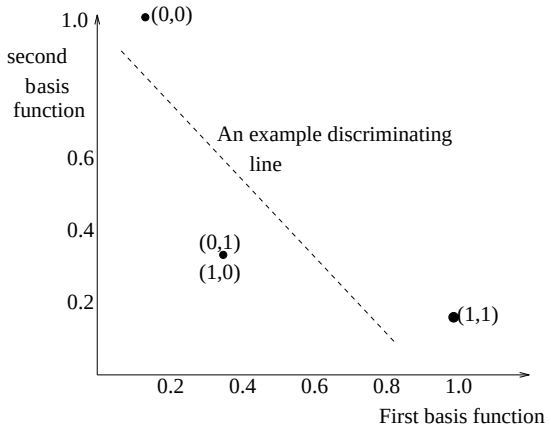

Figure 8.3: The hidden neurons' activations are plotted on a graph whose axes are the neuron's functions. One (of many possible) discriminating lines is drawn.

where, as before, D^k represents the target output for the k^{th} input pattern, then we can represent the instantaneous error (the error on presentation of a single pattern) by

$$E^k = \frac{1}{2}(D^k - \sum_j w_j \phi_j(\mathbf{x}^k))^2 = \frac{1}{2}(e^k)^2 \tag{8.3}$$

and so we can create an on-line learning algorithm using

$$\frac{\partial E^k}{\partial w_i} = -(D^k - \sum_j w_j \phi_j(\mathbf{x}^k))\phi_k(\mathbf{x}^k) = -e^k.\phi_i(\mathbf{x}^k) \tag{8.4}$$

Therefore we will change the weights after presentation of the k^{th} input pattern $\mathbf{x}^k$ by

$$\Delta w_i = -\frac{\partial E^k}{\partial w_i} = e^k.\phi_i(\mathbf{x}^k) \tag{8.5}$$

8.2.3 Approximation problems

We may view the problem of finding the weights which minimise the error at the outputs as an approximation problem. Then the learning process described above is equivalent to finding that line in the hidden layer's space which is optimal for approximating an unknown function. Note that this straight line (or in general hyperplane) in the hidden layer space is equivalent to a curve or hypersurface in the original space. Now, as before, our aim is not solely to make the best fit to the data points on which we are training the network; our overall aim is to have the network perform as well as possible on data which it has not seen during learning. Previously we described this as generalisation. In the context of the RBF network, we may here view it as interpolation: we are fitting the RBF network to the actual values which it sees during training but we are doing so with a smooth enough set of basis functions that we can interpolate between the training points and give correct (or almost correct) responses for points not in the training set.

If the number of points in the training set is less than the number of hidden neurons, this problem is underdetermined and there is the possibility that the hidden neurons will map the training data precisely and be less useful as an approximation function of the underlying distribution. Ideally the examples from which we wish to generalise must show some redundancy, however if this is not possible we can add some

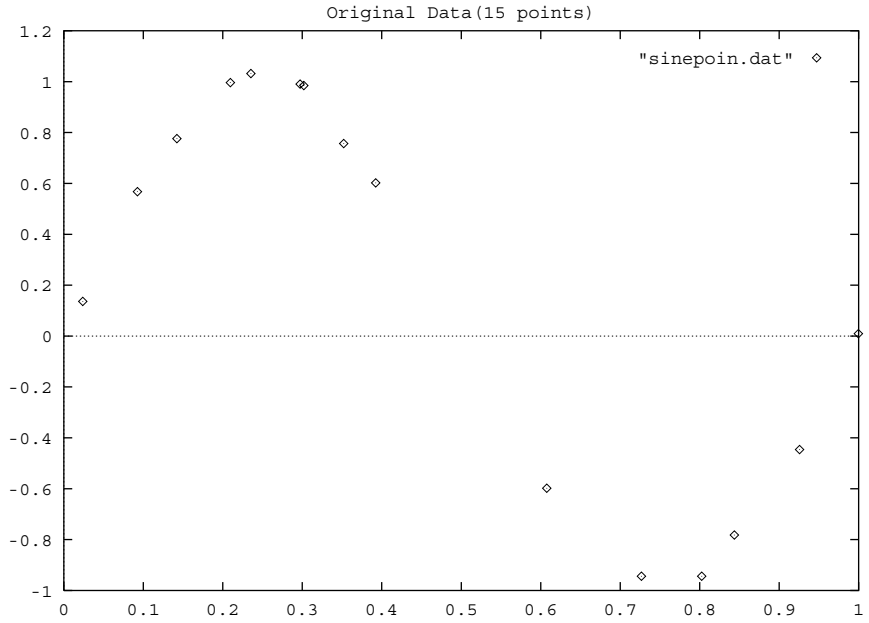

Figure 8.4: 15 data points drawn from a noisy version of $\sin(2\pi x)$.

constraints to the RBF network to attempt to ensure that any approximation it might perform is valid. A typical constraint is that the second derivative of the basis functions with respect to the input distribution is sufficiently small. If this is small, the weighted sum of the functions does not change too rapidly when the inputs change and so the output values (the y's) should be reasonable approximations of the true values of the unknown function at intermediate input values (the x's) between the training values.

8.2.4 RBF and MLP as Approximators

We examine the approximation properties of both a multi-layered perceptron and a radial basis function on a problem which we met in Chapter 4: a noisy version of a simple trigonometric function. Consider the set of points shown in Figure 8.4 which are drawn from $\sin(2\pi x)$ + noise. The convergence of radial basis function networks is shown in Figure 8.5. In all cases the centres of the basis functions were set evenly across the interval [0,1]. It is clear that the network with 1 basis function is not powerful enough to give a good approximation to the data. That with 3 basis functions makes a much better job while that with 5 is better yet. Note that in thelast cases the approximation near the end points (0 and 1) is much worse than that in the centre of the mapping. This illustrates the fact that RBFs are better at interpolation than extrapolation: where there is a region of the input space with little data, an RBF cannot be expected to approximate well.

The above results might suggest that we should simply create a large RBF network with a great many basis functions, however if we create too many basis functions the network will begin to model the noise rather than try to extract the underlying signal from the data. An example is shown in Figure 8.6. In order to compare the convergence of the RBF network with an MLP we repeat the experiment performed in Chapter 4 with the same data but with a multi-layered perceptron with linear output units and a tanh() nonlinearity in the hidden units. The results are shown in Figure 8.7.

Notice that in this case we were *required* to have biases on both the hidden neurons and the output

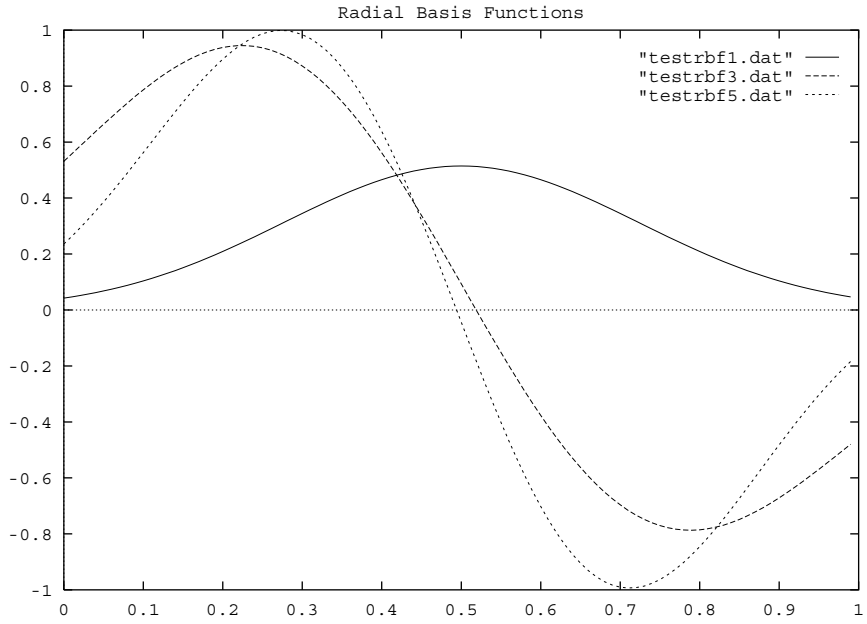

Figure 8.5: Approximation of the above data using radial basis function networks with 1, 3 and 5 basis functions.

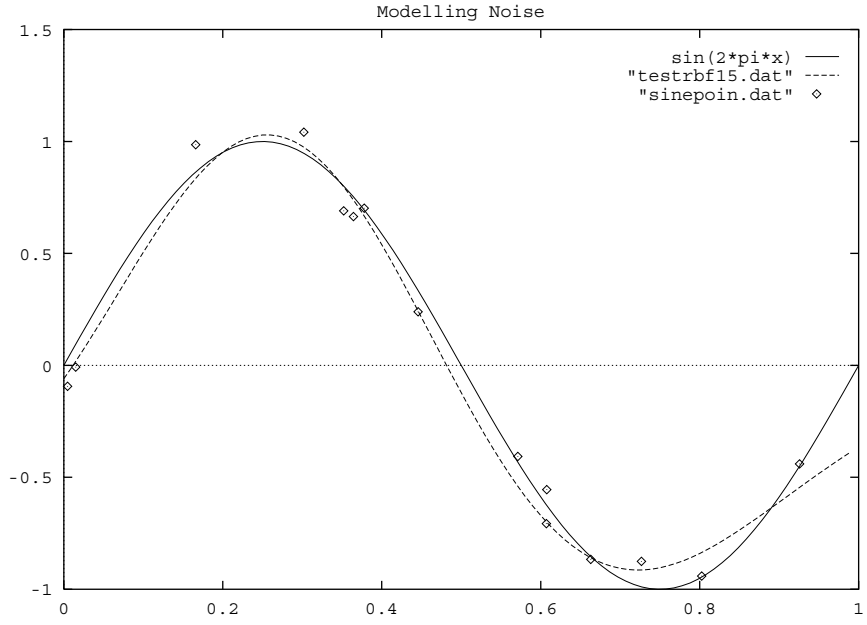

Figure 8.6: The data points which were corrupted by noise, the underlying signal and the network approximation by a radial basis function net with 15 basis functions.

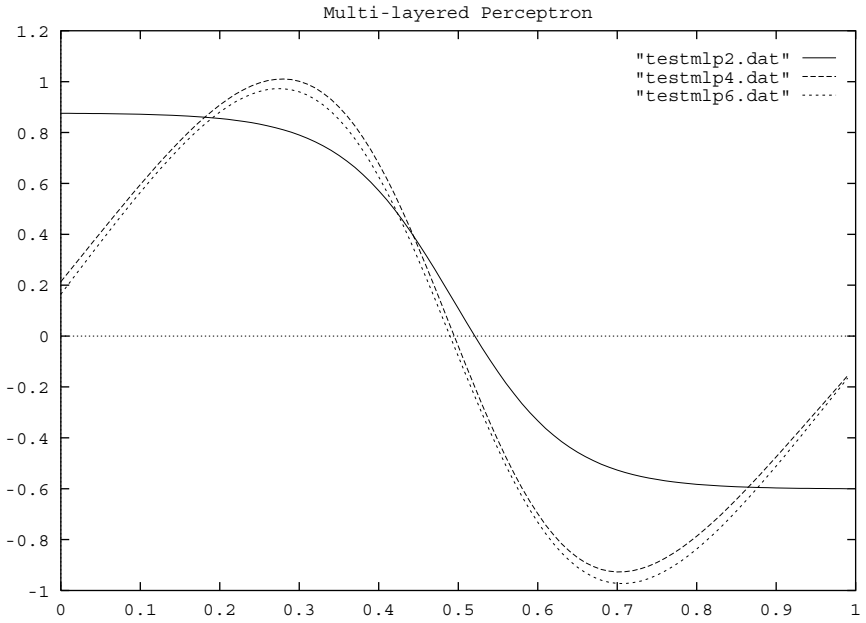

Figure 8.7: A comparison of the network convergence using multilayered perceptrons on the same data.

neurons and so the nets in the Figure had 1, 3 and 5 hidden neurons *plus* a bias neuron in each case. This is necessary because

- In an RBF network, activation contours (where the hidden neurons fire equally are circular (or ellipsoid if the function has a different response in each direction).
- In an MLP network, activation contours are planar - the hidden neurons have equal responses to a plane of input activations which must go through the origin if there is no bias.

However the number of basis neurons is not the only parameter in the RBF. We can also change its properties when we move the centres or change the width of the basis functions. We illustrate this last in Figure 8.8 in which we illustrate this fact on the same type of data as before but use a value of $\lambda = 1$ and $\lambda = 100$ for the parameter λ when calculating the output of the basis neurons.

$$y = \sum_i exp(-\lambda \parallel \mathbf{x}_i - \mathbf{c}_i \parallel^2) \tag{8.6}$$

8.2.5 Comparison with MLPs

Both RBFs and MLPs can be shown to be universal approximators i.e. each can arbitrarily closely model continuous functions. There are however several important differences:

1. The neurons of an MLP generally all calculate the same function of the neurons' activations e.g. all neurons calculate the logistic function of their weighted inputs. In an RBF, the hidden neurons perform a non-linear mapping whereas the output layer is always linear.

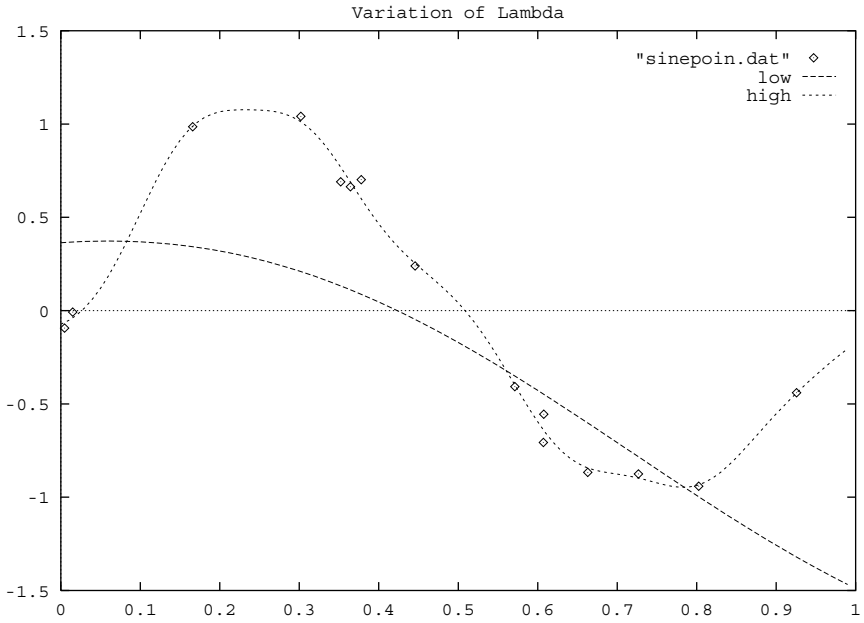

Figure 8.8: Using a basis function with a wide neighbourhood is equivalent to smoothing the output. A narrower neighbourhood function will more closely model the noise.

2. The non-linearity in MLPs is generally monotonic; in RBFs we use a radially decreasing function.

3. The argument of the MLP neuron's function is the vector product $\mathbf{w}.\mathbf{x}$ of the input and the weights; in an RBF network, the argument is the distance between the input and the centre of the radial basis function, $\| \mathbf{x} - \mathbf{c} \|$.

4. MLPs perform a global calculation whereas RBFs find a sum of local outputs. Therefore MLPs are better at finding answers in regions of the input space where there is little data in the training set. If accurate results are required over the whole training space, we may require many RBFs i.e. many hidden neurons in an RBF network. However because of the local nature of the model, RBFs are less sensitive to the order in which data is presented to them.

5. MLPs must pass the error back in order to change weights progressively. RBFs do not do this and so are much quicker to train.

8.2.6 Finding the Centres of the RBFs

If we have little data, we may have no option but to position the centres of our radial basis functions at the data points. However, as noted earlier, such problems may be ill-posed and lead to poor generalisation. If we have more training data, several solutions are possible:

1. Choose the centres of the basis functions randomly from the available training data.

2. Choose to allocate each point to a particular radial basis function (i.e. such that the greatest component of the hidden layer's activation comes from a particular neuron) according to the *k-nearest neighbours rule*. In this rule, a vote is taken among the k-nearest neighbours as to which neuron's centre they

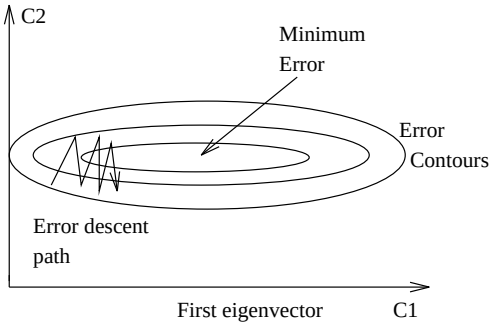

Figure 8.9: The path taken by the route of fastest descent is not the path directly to the centre (minimum) of the error surface.

are closest and the new input is allocated accordingly. The centre of the neuron is moved so that it remains the average of the inputs allocated to it.

3. We can use a generalisation of the LMS rule:

$$\Delta c_i = -\frac{\partial E}{\partial c_i} \tag{8.7}$$

This unfortunately is not guaranteed to converge (unlike the equivalent weight change rule) since the cost function E is not convex with respect to the centres and so a local minimum is possible.

8.3 Error Descent

The backpropagation method as described so far is innately slow. The reason for this is shown diagrammatically in Figure 8.9. In this Figure, we show (in two dimensions) the contours of constant error. Since the error is not the same in each direction we get elipses rather than circles. If we are following the path of steepest descent, which is perpendicular to the contours of constant error, we get a zig-zag path as shown. The axes of the elipse can be shown to be parallel to the eigenvectors of the Hessian matrix. The greater the difference between the largest and the smallest eigenvalues, the more eliptical the error surface is and the more zig-zag the path that the fastest descent algorithm takes.

8.3.1 Mathematical Background

The matrix of second derivatives is known as the Hessian and may be written as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_m} & \frac{\partial^2 E}{\partial w_2 \partial w_m} & \cdots & \frac{\partial^2 E}{\partial w_m^2} \end{bmatrix} \tag{8.8}$$

If we are in the neighbourhood of a minimum $\mathbf{w^*}$, we can consider the (truncated) Taylor series expansion of the error as

$$E(\mathbf{w}) = E(\mathbf{w}^*) + (\mathbf{w} - \mathbf{w}^*)^T \nabla \mathbf{E} + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) \tag{8.9}$$

where **H** is the Hessian matrix at **w*** and ∇**E** is the vector of derivatives of E at **w***. Now at the minimum (**w***), ∇**E** is zero and so we can approximate equation 8.9 with

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{8.10}$$

The eigenvectors of the Hessian,$\mathbf{c}_i$ are defined by

$$\mathbf{H}\mathbf{c}_i = \lambda_i \mathbf{c}_i \tag{8.11}$$

and form an orthonormal set (they are perpendicular to one another and have length 1) and so can be used as a basis of the **w** space. So we can write

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{c}_i \tag{8.12}$$

Now since $\mathbf{H}(\mathbf{w} - \mathbf{w}^*) = \sum_i \lambda_i \alpha_i \mathbf{c}_i$ we have

$$E(\mathbf{w}) = E(\mathbf{w}*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2 \tag{8.13}$$

In other words, the error is greatest in directions where the eigenvalues of the Hessian are greatest. Or alternatively, the contours of equal error are determined by $\frac{1}{\sqrt{\lambda_i}}$. So the long axis in Figure 8.9 has radius proportional to $\frac{1}{\sqrt{\lambda_1}}$ and the short axis has radius proportional to $\frac{1}{\sqrt{\lambda_2}}$. Ideally we would like to take this information into account when converging towards the minimum.

Now we have $\Delta E = \sum_i \alpha_i \lambda_i \mathbf{c}_i$ and we have $\Delta\mathbf{w} = \sum_i \Delta\alpha_i \mathbf{c}_i$ so since we wish to use $\Delta\mathbf{w} = -\eta\Delta E$, we have

$$\Delta\alpha_i = -\eta\lambda_i\alpha_i \tag{8.14}$$

and so

$$\alpha_i^{new} = (1 - \eta\lambda_i)\alpha_i^{old} \tag{8.15}$$

which gives us a means of adjusting the distance travelled along the eigenvector in each direction. So by taking a larger learning rate η we will converge quicker to the minimum error point in weight space. However, there are constraints in that the changes to α_i form a geometric sequence,

$$\begin{aligned}
\alpha_i^{(1)} &= (1 - \eta\lambda_i)\alpha_i^{(0)} \\
\alpha_i^{(2)} &= (1 - \eta\lambda_i)\alpha_i^{(1)} = (1 - \eta\lambda_i)^2\alpha_i^{(0)} \\
\alpha_i^{(3)} &= (1 - \eta\lambda_i)\alpha_i^{(2)} = (1 - \eta\lambda_i)^3\alpha_i^{(0)} \\
\alpha_i^{(T)} &= (1 - \eta\lambda_i)\alpha_i^{(T-1)} = (1 - \eta\lambda_i)^T\alpha_i^{(0)}
\end{aligned}$$

This will diverge if $|1 - \eta\lambda_i| > 1$. Therefore we must choose a value of η as large as possible but not so large as to break this bound. Therefore $\eta < \frac{2}{\lambda_1}$ where λ_1 is the greatest eigenvalue of the Hessian. But note that this means that the convergence along other directions will be at best proportional to $(1 - \frac{2\lambda_i}{\lambda_1})$ i.e. convergence is determined by the ratio of the smallest to the largest eigenvalues.

Thus gradient descent is *inherently* a slow method of finding the minimum of the error surface. We can now see the effect of momentum diagrammatically since the momentum is built up in direction $\mathbf{c}_1$ while the continual changes of sign in direction $\mathbf{c}_2$ causes little overall change in the momentum in this direction.

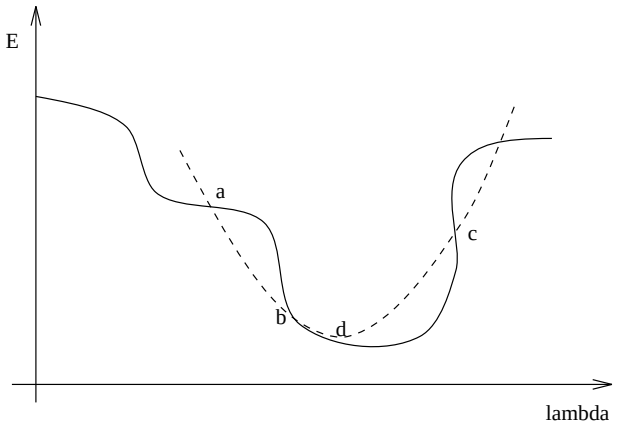

Figure 8.10: The error function as a function of λ. The (unknown) error function is shown as a solid line. The fitted parabola is shown as a dotted line with minimum value at d.

8.3.2 QuickProp

Fahlman has developed an heuristic which attempts to take into account the curvature of the error surface at any point by defining

$$\Delta w_{ij}(k) = \begin{cases} \alpha_{ij}(k)\Delta w_{ij}(k-1), & \text{if } \Delta w_{ij}(k-1) \neq 0 \\ \eta_0 \frac{\partial E}{\partial w_{ij}}, & \text{if } \Delta w_{ij}(k-1) = 0 \end{cases} \tag{8.16}$$

where

$$\alpha_{ij}(k) = \min(\frac{\frac{\partial E(k)}{\partial w_{ij}}}{\frac{\partial E(k-1)}{\partial w_{ij}} - \frac{\partial E(k)}{\partial w_{ij}}}, \alpha_{max}) \tag{8.17}$$

8.4 Line Search

If we know the direction in which we wish to go - the direction in which we will change the weights - we need only determine how far along the direction we wish to travel. We therefore choose a value of λ in

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda^{(t)}\mathbf{d}^{(t)} \tag{8.18}$$

in order to minimise

$$E(\lambda) = E(\mathbf{w}^{(t)} + \lambda^{(t)}\mathbf{d}^{(t)}) \tag{8.19}$$

We show E as a function of λ in Figure 8.10. If we start at a point a and have points b and c such that $E(a) > E(b)$ and $E(c) > E(b)$ then it follows that there must be a minimum between a and c. So we now fit a parabola to a, b and c and choose the minimum of that parabola to get d. Now we can choose 3 of these four points (one of which must be d) which also satisfy the above relation and iterate the parabola fitting and minimum finding.

8.4.1 Conjugate Gradients

Now we must find a method to find the direction $\mathbf{d}$ in which to search for the minimum. Our first attempt might be to find the best (minimum error) point along one direction and then start from there to find the

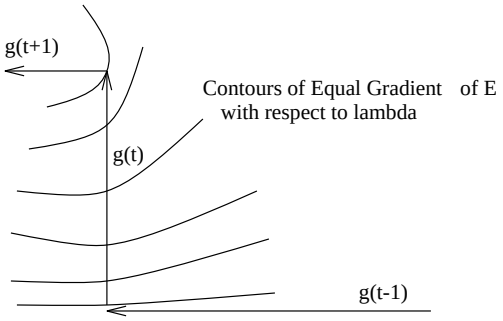

Figure 8.11: After minimising error in one direction, the new direction of fastest descent is perpendicular to that just traversed. This leads to a zigzagging approach to the minimum.

best point along the fastest descent direction from that point. However as shown in Figure 8.11, we see that this leads to zig-zagging and so the minimisation of the error function proceeds very slowly.

We require *conjugate* or non-interfering directions: we choose the direction $\mathbf{d}^{(t+1)}$ such that the component of the direction $\mathbf{d}^{(t)}$ is (approximately) unaltered.

The slowness of the vanilla backpropagation method is due partly at least to the interaction between different gradients. Thus the method zig-zags to the minimum of the error surface. The conjugate-gradient method avoids this problem by creating an intertwined relationship between the direction of change vector and the gradient vector. The method is

- Calculate the gradient vector $\mathbf{g}$ for the batch of patterns as usual. Call this $\mathbf{g}(0)$, the value of $\mathbf{g}$ at time 0 and let $\mathbf{p}(0) = \mathbf{g}(0)$.
- Update the weights according to
$$\Delta\mathbf{w}(n) = \eta(n)\mathbf{p}(n) \tag{8.20}$$
- Calculate the new gradient vector $\mathbf{g}$(n+1) with the usual method
- Calculate the parameter $\beta(n)$.
- Recalculate the new value of $\mathbf{p}$ using
$$\mathbf{p}(n+1) = -\mathbf{g}(n+1) + \beta(n)\mathbf{p}(n) \tag{8.21}$$
- Repeat from step 2 till convergence

The step left undefined is the calculation of the parameter $\beta(n)$. This can be done in a number of ways but the most common (and one of the easiest) is the Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}^T(n+1)\mathbf{g}(n+1)}{\mathbf{g}^T(n)\mathbf{g}(n)} \tag{8.22}$$

The calculation of the parameter $\eta(n)$ is done to minimise the cost function

$$E(\mathbf{w}(n) + \eta\mathbf{p}(n)) \tag{8.23}$$

As with the Newton method (see below), convergence using this method is much faster but computationally more expensive.

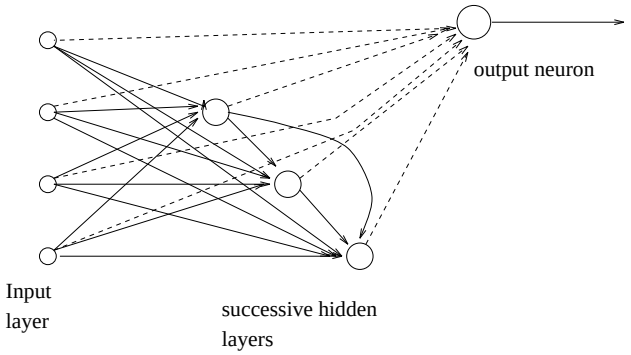

Figure 8.12: A cascade correlation network. Note that there are many hidden "layers" each containing only a single neuron. The dotted lines show weights which continue to be trainable during the whole simulation. The solid lines represent weights which are frozen after the initial learning period.

8.5 Cascade Correlation

One of the themes of our investigation of supervised learning networks has been the need to ensure that the networks perform well not just on the training set but also on data on which the network has not seen during training. We have shown that this can be achieved by cutting down on the number of parameters (weights) in the networks: if there are enough parameters, the network will simply act as a look-up table since it will have memorised the data.

Cascade correlation networks attempt to start off with a very simple network and then add neurons only when they are needed. A typical cascade correlation network is shown in figure 8.12. The algorithm is

1. Start with a network consisting of only an input and an output layer (in Figure 8.12, the output layer is only a single neuron).
2. Connect all input neurons to all output neurons.
3. Train the weights using the LMS algorithm till the error stops decreasing.
4. If the error is not small enough, generate a pool of "candidate neurons". Every candidate neuron is connected to all input neurons and all existing hidden neurons.
5. Try to maximise the correlation between the activation of each candidate neuron and the residual error in the network by training all the links leading to the candidate neurons. Stop training when the learning does not decrease the residual error.
6. Choose the candidate neuron with maximum correlation with the error at the outputs. Freeze its incoming weights and add it to the network creating a new link between it and the output neurons.
7. Loop back to 3 till the error is sufficiently small

It is important to note that the candidate neurons do not interact with each other and so they can be tested in isolation. To maximise the correlation (actually the covariance) between the candidate and the previous residual error, we first define

$$S = \sum_i |\sum_p (y^p - \langle y \rangle)(E_i^p - \langle E_i \rangle)| \tag{8.24}$$

where y^p is the output of the candidate neuron when the network is presented with the p^{th} pattern, $\langle y \rangle$ is its average value (taken over all input patterns), E_i^p is the residual error at the i^{th} output when the network

is presented with the p^{th} pattern and $\langle E_i \rangle$ is its average value. Then just as with other supervised learning methods we differentiate this with respect to the weights but since we wish to *maximise* S, we use

$$\Delta w_j = \frac{\partial S}{\partial w_j} = \sum_i \sum_p \sigma_i (E_i^p - \langle E_i \rangle) f'^p x_j^p \tag{8.25}$$

where σ_i is the sign of the correlation between the candidate's value and the residual error, f'^p is the derivative of the candidate neuron's activation function with respect to the sum of its inputs (for pattern p) and x_j^p is the input the candidate receives from unit j for pattern p.

The cascade correlation architecture takes many of the heuristics out of network creation. It learns fast since we do not have global learning taking place at any time. It will build deep nets in which, it is claimed, the hidden neurons act as high order feature detectors.

8.6 Reinforcement Learning

Reinforcement learning is sometimes distinguished from supervised learning since the teacher does not provide the correct answer to the current set of inputs. The teacher merely provides a training signal in terms of "right" or "wrong" to the network. Therefore the input data is presented to the network, activation is passed through the network just as in a multi-layered perceptron and the teacher then looks at the network's response to that input data. If the network has made the correct response, no action is taken; however, if the network has made an incorrect response, an error signal is sent to the network and it must adjust its weights to make the correct response more likely the next time.

This has been linked to biological learning [Haykin]:

> If an action taken by a learning system is followed by a satisfactory state of affairs, then the tendency of the system to produce that particular action is strengthened or reinforced. Otherwise, the tendency of the system to produce that action is weakened.

We could view GAs as an example of *nonassociative* reinforcement learning since the probability of reproducing may be thought of as the feedback signal from the environment. The field in which reinforcement learning has found its most prominent place is that of control. A typical problem is the pole-balancing experiment in which a vertical pole must be held in a vertical position using only the "bang-bang" backward/forward forces available to a simple cart on which the pole is balanced. A failure is easy to see while success is keeping the pole balanced for as long as possible. A second experiment is the truck-backing up experiment in which articulated vehicles must be reversed into a variety of parking positions. Such tasks take a prohibitively long time to learn using standard supervised learning techniques but have been learned using reinforcement learning.

Typical of reinforcement learning methods (and true of the two examples given above) is that the system is expected to probe its environment and learn from its mistakes. The pole starts of from an almost vertical position and the system attempts to prolong its success learning as it does so. Often the reward may be delayed thereby raising the **credit assignment problem** - how to decide at what stage a set of actions became less than useful.

8.7 Second Order Methods

The most obvious second order method is based on Newton's method. A feature of error descent is that if the slope (gradient of the error with respect to the weights) does not change much locally we can use a large learning rate while if the slope is very changeable we should use a small learning rate or face the danger of overshooting our minimum. Now the rate of change of the slope is the second derivative of the error with

respect to the weights. Thus if the second derivative is high we wish a small learning rate while if the second derivative is small we can use a high learning rate. The matrix of second derivatives is known as the Hessian and may be written as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_m} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_m} & \frac{\partial^2 E}{\partial w_2 \partial w_m} & \cdots & \frac{\partial^2 E}{\partial w_m^2} \end{bmatrix} \tag{8.26}$$

Therefore we use the weight update rule

$$\Delta \mathbf{w} \propto H^{-1} \frac{\partial E}{\partial \mathbf{w}} \tag{8.27}$$

This method is certain to cause faster convergence but the extra computational cost of calculating the Hessian at each iteration may make it infeasible as a practical proposition.

8.8 Exercises

1. The inverse multiquadric function

$$\phi(r) = \frac{1}{(r^2 + c^2)^{\frac{1}{2}}}, c > 0 \tag{8.28}$$

   has been suggested as a possible basis function where r is the distance from the centre of the function. Draw a 1 dimensional example of this function and discuss its properties as a basis function. (Objective 3).

2. Find another set of values (centres, weights) with which a radial basis function may solve the XOR problem. (Objective 3).

3. Compare the learning methods of this chapter with standard backprop. (Objectives 2,4).