Neural Networks: A Pattern Recognition Perspective[†]

Christopher M. Bishop

Neural Computing Research Group
Aston University, Birmingham, UK

January, 1996

Technical Report: NCRG/96/001
Available from: `http://www.ncrg.aston.ac.uk/`

1 Introduction

Neural networks have been exploited in a wide variety of applications, the majority of which are concerned with pattern recognition in one form or another. However, it has become widely acknowledged that the effective solution of all but the simplest of such problems requires a *principled* treatment, in other words one based on a sound theoretical framework.

From the perspective of pattern recognition, neural networks can be regarded as an extension of the many conventional techniques which have been developed over several decades. Lack of understanding of the basic principles of statistical pattern recognition lies at the heart of many of the common mistakes in the application of neural networks. In this chapter we aim to show that the 'black box' stigma of neural networks is largely unjustified, and that there is actually considerable insight available into the way in which neural networks operate, and how to use them effectively.

Some of the key points which are discussed in this chapter are as follows:

1. Neural networks can be viewed as a general framework for representing non-linear mappings between multi-dimensional spaces in which the form of the mapping is governed by a number of adjustable parameters. They therefore belong to a much larger class of such mappings, many of which have been studied extensively in other fields.

2. Simple techniques for representing multi-variate non-linear mappings in one or two dimensions (e.g. polynomials) rely on linear combinations of *fixed* basis functions (or 'hidden functions'). Such methods have severe limitations when extended to spaces of many dimensions; a phenomenon known as the *curse of dimensionality*. The key contribution of neural networks in this respect is that they employ basis functions which are themselves adapted to the data, leading to efficient techniques for multi-dimensional problems.

3. The formalism of statistical pattern recognition, introduced briefly in Section 2.3, lies at the heart of a principled treatment of neural networks. Many of these topics are treated in standard texts on statistical pattern recognition, including Duda and Hart (1973), Hand (1981), Devijver and Kittler (1982), and Fukunaga (1990).

[†]To be published in Fiesler E and Beale R (eds) 1996 Handbook of Neural Computation, (New York: Oxford University Press; Bristol: IOP Publishing Ltd)

4. Network training is usually based on the minimization of an error function. We show how error functions arise naturally from the principle of maximum likelihood, and how different choices of error function correspond to different assumptions about the statistical properties of the data. This allows the appropriate error function to be selected for a particular application.

5. The statistical view of neural networks motivates specific forms for the activation functions which arise in network models. In particular we see that the logistic sigmoid, often introduced by analogy with the mean firing rate of a biological neuron, is precisely the function which allows the activation of a unit to be given a particular probabilistic interpretation.

6. Provided the error function and activation functions are correctly chosen, the outputs of a trained network can be given precise interpretations. For regression problems they approximate the conditional averages of the distribution of target data, while for classification problems they approximate the posterior probabilities of class membership. This demonstrates why neural networks can approximate the optimal solution to a regression or classification problem.

7. Error back-propagation is introduced as a general framework for evaluating derivatives for feed-forward networks. The key feature of back-propagation is that it is computationally very efficient compared with a simple direct evaluation of derivatives. For network training algorithms, this efficiency is crucial.

8. The original learning algorithm for multi-layer feed-forward networks (Rumelhart *et al.*, 1986) was based on gradient descent. In fact the problem of optimizing the weights in a network corresponds to unconstrained non-linear optimization for which many substantially more powerful algorithms have been developed.

9. Network complexity, governed for example by the number of hidden units, plays a central role in determining the generalization performance of a trained network. This is illustrated using a simple curve fitting example in one dimension.

These and many related issues are discussed at greater length in Bishop (1995).

2 Classification and Regression

In this chapter we concentrate on the two most common kinds of pattern recognition problem. The first of these we shall refer to as *regression*, and is concerned with predicting the values of one or more continuous output variables, given the values of a number of input variables. Examples include prediction of the temperature of a plasma given values for the intensity of light emitted at various wavelengths, or the estimation of the fraction of oil in a multi-phase pipeline given measurements of the absorption of gamma beams along various cross-sectional paths through the pipe. If we denote the input variables by a vector $\mathbf{x}$ with components x_i where $i = 1, \ldots, d$ and the output variables by a vector $\mathbf{y}$ with components y_k where $k = 1, \ldots, c$ then the goal of the regression problem is to find a suitable set of functions which map the x_i to the y_k.

The second kind of task we shall consider is called *classification* and involves assigning input patterns to one of a set of discrete classes $\mathcal{C}_k$ where $k = 1, \ldots, c$. An important example involves the automatic interpretation of hand-written digits (Le Cun *et al.*, 1989). Again, we can formulate a classification problem in terms of a set of functions which map inputs x_i to outputs y_k where now the outputs specify which of the classes the input pattern belongs to. For instance, the input may be assigned to the class whose output value y_k is largest.

In general it will not be possible to determine a suitable form for the required mapping, except with the help of a data set of examples. The mapping is therefore modelled in terms of some mathematical function which contains a number of adjustable parameters, whose values are determined with the help of the data. We can write such functions in the form

$$y_k = y_k(\mathbf{x}; \mathbf{w}) \tag{1}$$

where $\mathbf{w}$ denotes the vector of parameters $w_1, \ldots, w_W$. A neural network model can be regarded simply as a particular choice for the set of functions $y_k(\mathbf{x}; \mathbf{w})$. In this case, the parameters comprising $\mathbf{w}$ are often called *weights*.

The importance of neural networks in this context is that they offer a very powerful and very general framework for representing non-linear mappings from several input variables to several output variables. The process of determining the values for these parameters on the basis of the data set is called *learning* or *training*, and for this reason the data set of examples is generally referred to as a *training set*. Neural network models, as well as many conventional approaches to statistical pattern recognition, can be viewed as specific choices for the functional forms used to represent the mapping (1), together with particular procedures for optimizing the parameters in the mapping. In fact, neural network models often contain conventional approaches (such as linear or logistic regression) as special cases.

2.1 Polynomial curve fitting

Many of the important issues concerning the application of neural networks can be introduced in the simpler context of curve fitting using polynomial functions. Here the problem is to fit a polynomial to a set of N data points by minimizing an error function. Consider the Mth-order polynomial given by

$$y(x) = w_0 + w_1 x + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j . \tag{2}$$

This can be regarded as a non-linear mapping which takes x as input and produces y as output. The precise form of the function $y(x)$ is determined by the values of the parameters $w_0, \ldots w_M$, which are analogous to the weights in a neural network. It is convenient to denote the set of parameters $(w_0, \ldots, w_M)$ by the vector $\mathbf{w}$ in which case the polynomial can be written as a functional mapping in the form (1). Values for the coefficients can be found by minimization of an error function, as will be discussed in detail in Section 3. We shall give some examples of polynomial curve fitting in Section 4

2.2 Why neural networks?

Pattern recognition problems, as we have already indicated, can be represented in terms of general parametrized non-linear mappings between a set of input variables and a set of output variables. A polynomial represents a particular class of mapping for the case of one input and one output. Provided we have a sufficiently large number of terms in the polynomial, we can approximate a wide class of functions to arbitrary accuracy. This suggests that we could simply extend the concept of a polynomial to higher dimensions. Thus, for d input variables, and again one output variable, we could, for instance, consider a third-order polynomial of the form

$$y = w_0 + \sum_{i_1=1}^{d} w_{i_1} x_{i_1} + \sum_{i_1=1}^{d} \sum_{i_2=1}^{d} w_{i_1 i_2} x_{i_1} x_{i_2} + \sum_{i_1=1}^{d} \sum_{i_2=1}^{d} \sum_{i_3=1}^{d} w_{i_1 i_2 i_3} x_{i_1} x_{i_2} x_{i_3} . \tag{3}$$

For an Mth-order polynomial of this kind, the number of independent adjustable parameters would grow like d^M, which represents a dramatic growth in the number of degrees of freedom in the model as the dimensionality of the input space increases. This is an example of the *curse of dimensionality* (Bellman, 1961). The presence of a large number of adaptive parameters in a model can cause major problems as we shall discuss in Section 4. In order that the model make good predictions for new inputs it is necessary that the number of data points in the training set be much greater than the number of adaptive parameters. For medium to large applications, such a model would need huge quantities of training data in order to ensure that the parameters (in this case the coefficients in the polynomial) were well determined.

There are in fact many different ways in which to represent general non-linear mappings between multidimensional spaces. The importance of neural networks, and similar techniques, lies in

the way in which they deal with the problem of scaling with dimensionality. In order to motivate neural network models it is convenient to represent the non-linear mapping function (1) in terms of a linear combination of *basis* functions, sometimes also called 'hidden functions' or *hidden units*, $z_j(\mathbf{x})$, so that

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj} z_j(\mathbf{x}). \tag{4}$$

Here the basis function z_0 takes the fixed value 1 and allows a constant term in the expansion. The corresponding weight parameter w_{k0} is generally called a *bias*. Both the one-dimensional polynomial (2) and the multi-dimensional polynomial (3) can be cast in this form, in which basis functions are fixed functions of the input variables.

We have seen from the example of the higher-order polynomial that to represent general functions of many input variables we have to consider a large number of basis functions, which in turn implies a large number of adaptive parameters. In most practical applications there will be significant correlations between the input variables so that the effective dimensionality of the space occupied by the data (known as the *intrinsic dimensionality*) is significantly less than the number of inputs. The key to constructing a model which can take advantage of this phenomenon is to allow the basis functions themselves to be adapted to the data as part of the training process. In this case the number of such functions only needs to grow as the complexity of the problem itself grows, and not simply as the number of input variables grows. The number of free parameters in such models, for a given number of hidden functions, typically only grows linearly (or quadratically) with the dimensionality of the input space, as compared with the d^M growth for a general Mth-order polynomial.

One of the simplest, and most commonly encountered, models with adaptive basis functions is given by the two-layer feed-forward network, sometimes called a *multi-layer perceptron*, which can be expressed in the form (4) in which the basis functions themselves contain adaptive parameters and are given by

$$z_j(\mathbf{x}) = g\left(\sum_{i=0}^{d} w_{ji} x_i\right) \tag{5}$$

where w_{j0} are bias parameters, and we have introduced an extra 'input variable' $x_0 = 1$ in order to allow the biases to be treated on the same footing as the other parameters and hence be absorbed into the summation in (5). The function $g(\cdot)$ is called an *activation function* and must be a non-linear function of its argument in order that the network model can have general approximation capabilities. If $g(\cdot)$ were linear, then (5) would reduce to the composition of two linear mappings which would itself be linear. The activation function is also chosen to be a differentiable function of its argument in order that the network parameters can be optimized using gradient-based methods as discussed in Section 3.3. Many different forms of activation function can be considered. However, the most common are sigmoidal (meaning 'S-shaped') and include the logistic sigmoid

$$g(a) = \frac{1}{1 + \exp(-a)} \tag{6}$$

which is plotted in Figure 1. The motivation for this form of activation function is considered in Section 3.2. We can combine (4) and (5) to obtain a complete expression for the function represented by a two-layer feed-forward network in the form

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj} g\left(\sum_{i=0}^{d} w_{ji} x_i\right). \tag{7}$$

The form of network mapping given by (7) is appropriate for regression problems, but needs some modification for classification applications as will also be discussed in Section 3.2.

It should be noted that models of this kind, with basis functions which are adapted to the data, are not unique to neural networks. Such models have been considered for many years in

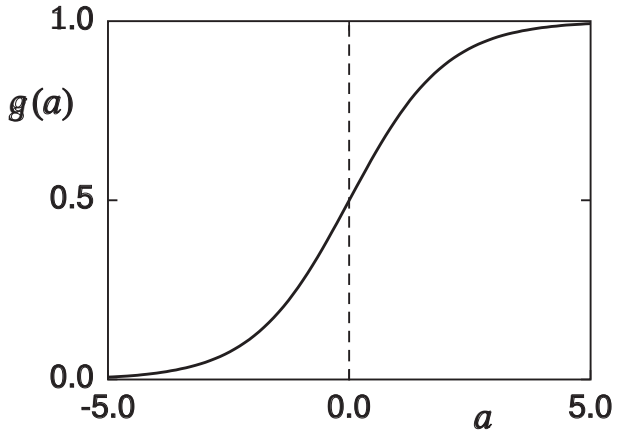

Figure 1. Plot of the logistic sigmoid activation function given by (6).

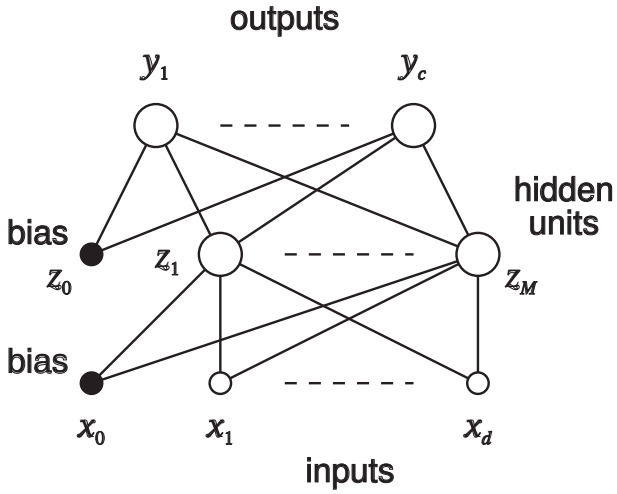

Figure 2. An example of a feed-forward network having two layers of adaptive weights.

the statistics literature and include, for example, *projection pursuit regression* (Friedman and Stuetzle, 1981; Huber, 1985) which has a form remarkably similar to that of the feed-forward network discussed above. The procedures for determining the parameters in projection pursuit regression are, however, quite different from those generally used for feed-forward networks.

It is often useful to represent the network mapping function in terms of a network diagram, as shown in Figure 2. Each element of the diagram represents one of the terms of the corresponding mathematical expression. The bias parameters in the first layer are shown as weights from an extra input having a fixed value of $x_0 = 1$. Similarly, the bias parameters in the second layer are shown as weights from an extra hidden unit, with activation again fixed at $z_0 = 1$.

More complex forms of feed-forward network function can be considered, corresponding to more complex topologies of network diagram. However, the simple structure of Figure 2 has the property that it can approximate any continuous mapping to arbitrary accuracy provided the number M of hidden units is sufficiently large. This property has been discussed by many authors including Funahashi (1989), Hecht-Nielsen (1989), Cybenko (1989), Hornik *et al.* (1989), Stinchecombe and White (1989), Cotter (1990), Ito (1991), Hornik (1991) and Kreinovich (1991). A proof that two-layer networks having sigmoidal hidden units can simultaneously approximate both a function and its derivatives was given by Hornik *et al.* (1990).

The other major class of network model, which also possesses universal approximation capabilities, is the *radial basis function network* (Broomhead and Lowe, 1988; Moody and Darken, 1989). Such networks again take the form (4), but the basis functions now depend on some measure of *distance* between the input vector $\mathbf{x}$ and a prototype vector $\boldsymbol{\mu}_j$. A typical example would be a Gaussian basis function of the form

$$z_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right) \tag{8}$$

where the parameter σ_j controls the width of the basis function. Training of radial basis function networks usually involves a two-stage procedure in which the basis functions are first optimized using input data alone, and then the parameters w_{kj} in (4) are optimized by error function minimization. Such procedures are described in detail in Bishop (1995).

2.3 Statistical pattern recognition

We turn now to some of the formalism of statistical pattern recognition, which we regard as essential for a clear understanding of neural networks. For convenience we introduce many of the central concepts in the context of classification problems, although much the same ideas apply also to regression. The goal is to assign an input pattern $\mathbf{x}$ to one of c classes $\mathcal{C}_k$ where $k = 1, \ldots, c$. In the case of hand-written digit recognition, for example, we might have ten classes corresponding to the ten digits $0, \ldots, 9$. One of the powerful results of the theory of statistical pattern recognition is a formalism which describes the theoretically best achievable performance, corresponding to the smallest probability of misclassifying a new input pattern. This provides a principled context within which we can develop neural networks, and other techniques, for classification.

For any but the simplest of classification problems it will not be possible to devise a system which is able to give perfect classification of all possible input patterns. The problem arises because many input patterns cannot be assigned unambiguously to one particular class. Instead the most general description we can give is in terms of the probabilities of belonging to each of the classes $\mathcal{C}_k$ *given* an input vector $\mathbf{x}$. These probabilities are written as $P(\mathcal{C}_k|\mathbf{x})$, and are called the *posterior* probabilities of class membership, since they correspond to the probabilities after we have observed the input pattern $\mathbf{x}$. If we consider a large set of patterns all from a particular class $\mathcal{C}_k$ then we can consider the probability distribution of the corresponding input patterns, which we write as $p(\mathbf{x}|\mathcal{C}_k)$. These are called the class-conditional distributions and, since the vector $\mathbf{x}$ is a continuous variable, they correspond to probability density functions rather than probabilities. The distribution of input vectors, irrespective of their class labels, is written as $p(\mathbf{x})$ and is called the *unconditional* distribution of inputs. Finally, we can consider the probabilities of occurrence of the different classes irrespective of the input pattern, which we write as $P(\mathcal{C}_k)$. These correspond to the relative frequencies of patterns within the complete data set, and are called *prior* probabilities since they correspond to the probabilities of membership of each of the classes before we observe a particular input vector.

These various probabilities can be related using two standard results from probability theory. The first is the *product rule* which takes the form

$$P(\mathcal{C}_k, \mathbf{x}) = P(\mathcal{C}_k|\mathbf{x})p(\mathbf{x}) \tag{9}$$

and the second is the *sum rule* given by

$$\sum_k P(\mathcal{C}_k, \mathbf{x}) = p(\mathbf{x}). \tag{10}$$

From these rules we obtain the following relation

$$P(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k)}{p(\mathbf{x})} \tag{11}$$

which is known as *Bayes' theorem*. The denominator in (11) is given by

$$p(\mathbf{x}) = \sum_k p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k) \tag{12}$$

and plays the role of a normalizing factor, ensuring that the posterior probabilities in (11) sum to one $\sum_k P(\mathcal{C}_k|\mathbf{x}) = 1$. As we shall see shortly, knowledge of the posterior probabilities allows us to find the optimal solution to a classification problem. A key result, discussed in Section 3.2, is that under suitable circumstances the outputs of a correctly trained neural network can be interpreted as (approximations to) the posterior probabilities $P(\mathcal{C}_k|\mathbf{x})$ when the vector $\mathbf{x}$ is presented to the inputs of the network.

As we have already noted, perfect classification of all possible input vectors will in general be impossible. The best we can do is to minimize the probability that an input will be misclassified. This is achieved by assigning each new input vector $\mathbf{x}$ to that class for which the posterior probability $P(\mathcal{C}_k|\mathbf{x})$ is largest. Thus an input vector $\mathbf{x}$ is assigned to class $\mathcal{C}_k$ if

$$P(\mathcal{C}_k|\mathbf{x}) > P(\mathcal{C}_j|\mathbf{x}) \qquad \text{for all } j \neq k. \tag{13}$$

We shall see the justification for this rule shortly. Since the denominator in Bayes' theorem (11) is independent of the class, we see that this is equivalent to assigning input patterns to class $\mathcal{C}_k$ provided

$$p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k) > p(\mathbf{x}|\mathcal{C}_j)P(\mathcal{C}_j) \qquad \text{for all } j \neq k. \tag{14}$$

A pattern classifier provides a rule for assigning each point of feature space to one of c classes. We can therefore regard the feature space as being divided up into c *decision regions* $\mathcal{R}_1, \ldots, \mathcal{R}_c$ such that a point falling in region $\mathcal{R}_k$ is assigned to class $\mathcal{C}_k$. Note that each of these regions need not be contiguous, but may itself be divided into several disjoint regions all of which are associated with the same class. The boundaries between these regions are known as *decision surfaces* or *decision boundaries*.

In order to find the optimal criterion for placement of decision boundaries, consider the case of a one-dimensional feature space x and two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. We seek a decision boundary which minimizes the probability of misclassification, as illustrated in Figure 3. A misclassification error will occur if we assign a new pattern to class $\mathcal{C}_1$ when in fact it belongs to class $\mathcal{C}_2$, or vice versa. We can calculate the total probability of an error of either kind by writing (Duda and Hart, 1973)

$$\begin{aligned} P(\text{error}) &= P(x \in \mathcal{R}_2, \mathcal{C}_1) + P(x \in \mathcal{R}_1, \mathcal{C}_2) \\ &= P(x \in \mathcal{R}_2|\mathcal{C}_1)P(\mathcal{C}_1) + P(x \in \mathcal{R}_1|\mathcal{C}_2)P(\mathcal{C}_2) \\ &= \int_{\mathcal{R}_2} p(x|\mathcal{C}_1)P(\mathcal{C}_1)\,dx + \int_{\mathcal{R}_1} p(x|\mathcal{C}_2)P(\mathcal{C}_2)\,dx \end{aligned} \tag{15}$$

where $P(x \in \mathcal{R}_1, \mathcal{C}_2)$ is the joint probability of x being assigned to class $\mathcal{C}_1$ and the true class being $\mathcal{C}_2$. From (15) we see that, if $p(x|\mathcal{C}_1)P(\mathcal{C}_1) > p(x|\mathcal{C}_2)P(\mathcal{C}_2)$ for a given x, we should choose the regions $\mathcal{R}_1$ and $\mathcal{R}_2$ such that x is in $\mathcal{R}_1$, since this gives a smaller contribution to the error. We recognise this as the decision rule given by (14) for minimizing the probability of misclassification. The same result can be seen graphically in Figure 3, in which misclassification errors arise from the shaded region. By choosing the decision boundary to coincide with the value of x at which the two distributions cross (shown by the arrow) we minimize the area of the shaded region and hence minimize the probability of misclassification. This corresponds to classifying each new pattern x using (14), which is equivalent to assigning each pattern to the class having the largest posterior probability. A similar justification for this decision rule may be given for the general case of c classes and d-dimensional feature vectors (Duda and Hart, 1973).

It is important to distinguish between two separate stages in the classification process. The first is *inference* whereby data is used to determine values for the posterior probabilities. These are then used in the second stage which is *decision making* in which those probabilities are used to make decisions such as assigning a new data point to one of the possible classes. So far we

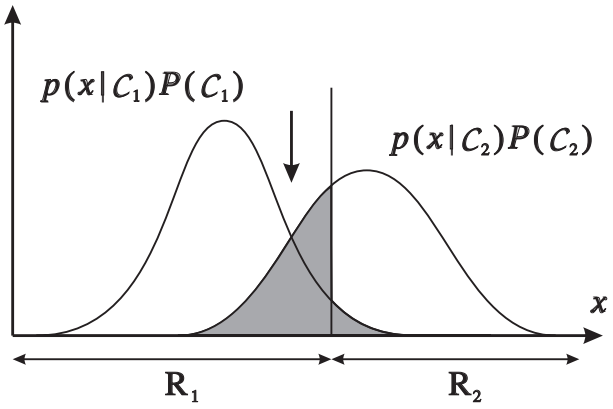

Figure 3. Schematic illustration of the joint probability densities, given by $p(x, \mathcal{C}_k) = p(x|\mathcal{C}_k)P(\mathcal{C}_k)$, as a function of a feature value x, for two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. If the vertical line is used as the decision boundary then the classification errors arise from the shaded region. By placing the decision boundary at the point where the two probability density curves cross (shown by the arrow), the probability of misclassification is minimized.

have based classification decisions on the goal of minimizing the probability of misclassification. In many applications this may not be the most appropriate criterion. Consider, for instance, the task of classifying images used in medical screening into two classes corresponding to 'normal' and 'tumour'. There may be much more serious consequences if we classify an image of a tumour as normal than if we classify a normal image as that of a tumour. Such effects may easily be taken into account by the introduction of a *loss matrix* with elements L_{kj} specifying the penalty associated with assigning a pattern to class $\mathcal{C}_j$ when in fact it belongs to class $\mathcal{C}_k$. The overall expected loss is minimized if, for each input $\mathbf{x}$, the decision regions $\mathcal{R}_j$ are chosen such that $\mathbf{x} \in \mathcal{R}_j$ when

$$\sum_{k=1}^{c} L_{kj}\, p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k) < \sum_{k=1}^{c} L_{ki} p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k) \qquad \text{for all } i \neq j \tag{16}$$

which represents a generalization of the usual decision rule for minimizing the probability of misclassification. Note that, if we assign a loss of 1 if the pattern is placed in the wrong class, and a loss of 0 if it is placed in the correct class, so that $L_{kj} = 1 - \delta_{kj}$ (where δ_{kj} is the Kronecker delta symbol), then (16) reduces to the decision rule for minimizing the probability of misclassification, given by (14).

Another powerful consequence of knowing posterior probabilities is that it becomes possible to introduce a *reject criterion*. In general we expect most of the misclassification errors to occur in those regions of $\mathbf{x}$-space where the largest of the posterior probabilities is relatively low, since there is then a strong overlap between different classes. In some applications it may be better not to make a classification decision in such cases. This leads to the following procedure

$$\text{if} \quad \max_k P(\mathcal{C}_k|\mathbf{x}) \quad \begin{cases} \geq \theta, & \text{then classify } \mathbf{x} \\ < \theta, & \text{then reject } \mathbf{x} \end{cases} \tag{17}$$

where θ is a threshold in the range $(0, 1)$. The larger the value of θ, the fewer points will be classified. For the medical classification problem for example, it may be better not to rely on an automatic classification system in doubtful cases, but to have these classified instead by a human expert.

Yet another application for the posterior probabilities arises when the distributions of patterns between the classes, corresponding to the prior probabilities $P(\mathcal{C}_k)$, are strongly mis-matched. If we know the posterior probabilities corresponding to the data in the training set, it is then it is a simple matter to use Bayes' theorem (11) to make the necessary corrections. This is achieved

by dividing the posterior probabilities by the prior probabilities corresponding to the training set, multiplying them by the new prior probabilities, and then normalizing the results. Changes in the prior probabilities can therefore be accommodated without retraining the network. The prior probabilities for the training set may be estimated simply by evaluating the fraction of the training set data points in each class. Prior probabilities corresponding to the operating environment can often be obtained very straightforwardly since only the class labels are needed and no input data is required. As an example, consider again the problem of classifying medical images into 'normal' and 'tumour'. When used for screening purposes, we would expect a very small prior probability of 'tumour'. To obtain a good variety of tumour images in the training set would therefore require huge numbers of training examples. An alternative is to increase artificially the proportion of tumour images in the training set, and then to compensate for the different priors on the test data as described above. The prior probabilities for tumours in the general population can be obtained from medical statistics, without having to collect the corresponding images. Correction of the network outputs is then a simple matter of multiplication and division.

The most common approach to the use of neural networks for classification involves having the network itself directly produce the classification decision. As we have seen, knowledge of the posterior probabilities is substantially more powerful.

3 Error Functions

We turn next to the problem of determining suitable values for the weight parameters $\mathbf{w}$ in a network.

Training data is provided in the form of N pairs of input vectors $\mathbf{x}^n$ and corresponding desired output vectors $\mathbf{t}^n$ where $n = 1, \ldots, N$ labels the patterns. These desired outputs are called *target* values in the neural network context, and the components t_k^n of $\mathbf{t}^n$ represent the targets for the corresponding network outputs y_k. For associative prediction problems of the kind we are considering, the most general and complete description of the statistical properties of the data is given in terms of the conditional density of the target data $p(\mathbf{t}|\mathbf{x})$ conditioned on the input data.

A principled way to devise an error function is to use the concept of *maximum likelihood*. For a set of training data $\{\mathbf{x}^n, \mathbf{t}^n\}$, the likelihood can be written as

$$\mathcal{L} = \prod_n p(\mathbf{t}^n|\mathbf{x}^n) \tag{18}$$

where we have assumed that each data point $(\mathbf{x}^n, \mathbf{t}^n)$ is drawn independently from the same distribution, so that the likelihood for the complete data set is given by the product of the probabilities for each data point separately. Instead of maximizing the likelihood, it is generally more convenient to minimize the negative logarithm of the likelihood. These are equivalent procedures, since the negative logarithm is a monotonic function. We therefore minimize

$$E = -\ln \mathcal{L} = -\sum_n \ln p(\mathbf{t}^n|\mathbf{x}^n) \tag{19}$$

where E is called an *error function*. We shall further assume that the distribution of the individual target variables t_k, where $k = 1, \ldots, c$, are independent, so that we can write

$$p(\mathbf{t}|\mathbf{x}) = \prod_{k=1}^{c} p(t_k|\mathbf{x}). \tag{20}$$

As we shall see, a feed-forward neural network can be regarded as a framework for modelling the conditional probability density $p(\mathbf{t}|\mathbf{x})$. Different choices of error function then arise from different assumptions about the form of the conditional distribution $p(\mathbf{t}|\mathbf{x})$. It is convenient to discuss error functions for regression and classification problems separately.

3.1 Error functions for regression

For regression problems, the output variables are continuous. To define a specific error function we must make some choice for the model of the distribution of target data. The simplest assumption is to take this distribution to be Gaussian. More specifically, we assume that the target variable t_k is given by some deterministic function of $\mathbf{x}$ with added Gaussian noise ϵ, so that

$$t_k = h_k(\mathbf{x}) + \epsilon_k. \tag{21}$$

We then assume that the errors ϵ_k have a normal distribution with zero mean, and standard a deviation σ which does not depend on $\mathbf{x}$ or k. Thus, the distribution of ϵ_k is given by

$$p(\epsilon_k) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{\epsilon_k^2}{2\sigma^2}\right). \tag{22}$$

We now model the functions $h_k(\mathbf{x})$ by a neural network with outputs $y_k(\mathbf{x};\mathbf{w})$ where $\mathbf{w}$ is the set of weight parameters governing the neural network mapping. Using (21) and (22) we see that the probability distribution of target variables is given by

$$p(t_k|\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{\{y_k(\mathbf{x};\mathbf{w}) - t_k\}^2}{2\sigma^2}\right) \tag{23}$$

where we have replaced the unknown function $h_k(\mathbf{x})$ by our model $y_k(\mathbf{x};\mathbf{w})$. Together with (19) and (20) this leads to the following expression for the error function

$$E = \frac{1}{2\sigma^2} \sum_{n=1}^{N} \sum_{k=1}^{c} \{y_k(\mathbf{x}^n;\mathbf{w}) - t_k^n\}^2 + Nc \ln\sigma + \frac{Nc}{2} \ln(2\pi). \tag{24}$$

We note that, for the purposes of error minimization, the second and third terms on the right-hand side of (24) are independent of the weights $\mathbf{w}$ and so can be omitted. Similarly, the overall factor of $1/\sigma^2$ in the first term can also be omitted. We then finally obtain the familiar expression for the sum-of-squares error function

$$E = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}^n;\mathbf{w}) - \mathbf{t}^n\|^2. \tag{25}$$

Note that models of the form (4), with fixed basis functions, are linear functions of the parameters $\mathbf{w}$ and so (25) is a quadratic function of $\mathbf{w}$. This means that the minimum of E can be found in terms of the solution of a set of linear algebraic equations. For this reason, the process of determining the parameters in such models is extremely fast. Functions which depend linearly on the adaptive parameters are called *linear* models, even though they may be non-linear functions of the input variables. If the basis functions themselves contain adaptive parameters, we have to address the problem of minimizing an error function which is generally highly non-linear.

The sum-of-squares error function was derived from the requirement that the network output vector should represent the conditional mean of the target data, as a function of the input vector. It is easily shown (Bishop, 1995) that minimization of this error, for an infinitely large data set and a highly flexible network model, does indeed lead to a network satisfying this property.

We have derived the sum-of-squares error function on the assumption that the distribution of the target data is Gaussian. For some applications, such an assumption may be far from valid (if the distribution is multi-modal for instance) in which case the use of a sum-of-squares error function can lead to extremely poor results. Examples of such distributions arise frequently in inverse problems such as robot kinematics, the determination of spectral line parameters from the spectrum itself, or the reconstruction of spatial data from line-of-sight information. One general approach in such cases is to combine a feed-forward network with a *Gaussian mixture model* (i.e. a linear combination of Gaussian functions) thereby allowing general conditional distributions $p(\mathbf{t}|\mathbf{x})$ to be modelled (Bishop, 1994).

3.2 Error functions for classification

In the case of classification problems, the goal as we have seen is to approximate the posterior probabilities of class membership $P(\mathcal{C}_k|\mathbf{x})$ given the input pattern $\mathbf{x}$. We now show how to arrange for the outputs of a network to approximate these probabilities.

First we consider the case of two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. In this case we can consider a network having a singly output y which we should represent the posterior probability $P(\mathcal{C}_1|\mathbf{x})$ for class $\mathcal{C}_1$. The posterior probability of class $\mathcal{C}_2$ will then be given by $P(\mathcal{C}_2|\mathbf{x}) = 1 - y$. To achieve this we consider a target coding scheme for which $t = 1$ if the input vector belongs to class $\mathcal{C}_1$ and $t = 0$ if it belongs to class $\mathcal{C}_2$. We can combine these into a single expression, so that the probability of observing either target value is

$$p(t|\mathbf{x}) = y^t(1-y)^{1-t} \tag{26}$$

which is a particular case of the binomial distribution called the Bernoulli distribution. With this interpretation of the output unit activations, the likelihood of observing the training data set, assuming the data points are drawn independently from this distribution, is then given by

$$\prod_n (y^n)^{t^n}(1-y^n)^{1-t^n}. \tag{27}$$

As usual, it is more convenient to minimize the negative logarithm of the likelihood. This leads to the *cross-entropy* error function (Hopfield, 1987; Baum and Wilczek, 1988; Solla *et al.*, 1988; Hinton, 1989; Hampshire and Pearlmutter, 1990) in the form

$$E = -\sum_n \left\{ t^n \ln y^n + (1-t^n)\ln(1-y^n) \right\}. \tag{28}$$

For the network model introduced in (4) the outputs were linear functions of the activations of the hidden units. While this is appropriate for regression problems, we need to consider the correct choice of output unit activation function for the case of classification problems. We shall assume (Rumelhart *et al.*, 1995) that the class-conditional distributions of the outputs of the hidden units, represented here by the vector $\mathbf{z}$, are described by

$$p(\mathbf{z}|\mathcal{C}_k) = \exp\left\{ A(\boldsymbol{\theta}_k) + B(\mathbf{z}, \boldsymbol{\phi}) + \boldsymbol{\theta}_k^{\mathrm{T}}\mathbf{z} \right\} \tag{29}$$

which is a member of the *exponential family* of distributions (which includes many of the common distributions as special cases such as Gaussian, binomial, Bernoulli, Poisson, and so on). The parameters $\boldsymbol{\theta}_k$ and $\boldsymbol{\phi}$ control the form of the distribution. In writing (29) we are implicitly assuming that the distributions differ only in the parameters $\boldsymbol{\theta}_k$ and not in $\boldsymbol{\phi}$. An example would be two Gaussian distributions with different means, but with common covariance matrices. (Note that the decision boundaries will then be linear functions of $\mathbf{z}$ but will of course be non-linear functions of the input variables as a consequence of the non-linear transformation by the hidden units).

Using Bayes' theorem, we can write the posterior probability for class $\mathcal{C}_1$ in the form

$$\begin{aligned} P(\mathcal{C}_1|\mathbf{z}) &= \frac{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1) + p(\mathbf{z}|\mathcal{C}_2)P(\mathcal{C}_2)} \\ &= \frac{1}{1+\exp(-a)} \end{aligned} \tag{30}$$

which is a logistic sigmoid function, in which

$$a = \ln \frac{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{z}|\mathcal{C}_2)P(\mathcal{C}_2)} \tag{31}$$

Using (29) we can write this in the form

$$a = \mathbf{w}^{\mathrm{T}}\mathbf{z} + w_0 \tag{32}$$

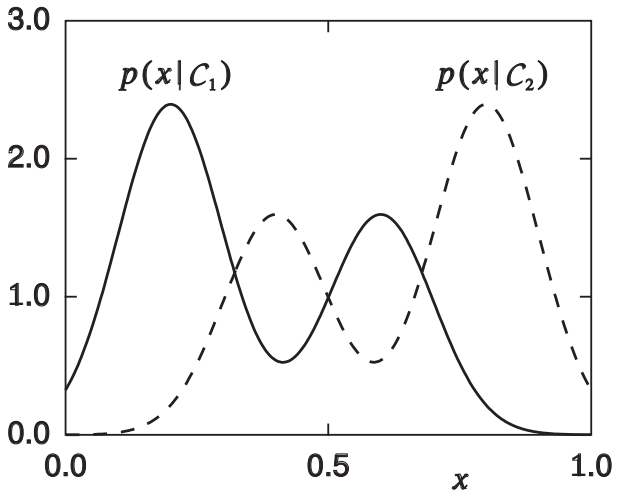

Figure 4. Plots of the class-conditional densities used to generate a data set to demonstrate the interpretation of network outputs as posterior probabilities. The training data set was generated from these densities, using equal prior probabilities.

where we have defined

$$\mathbf{w} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_2 \tag{33}$$

$$w_0 = A(\boldsymbol{\theta}_1) - A(\boldsymbol{\theta}_2) + \ln \frac{P(\mathcal{C}_1)}{P(\mathcal{C}_2)}. \tag{34}$$

Thus the network output is given by a logistic sigmoid activation function acting on a weighted linear combination of the outputs of those hidden units which send connections to the output unit.

Incidentally, it is clear that we can also apply the above arguments to the activations of hidden units in a network. Provided such units use logistic sigmoid activation functions, we can interpret their outputs as probabilities of the presence of corresponding 'features' conditioned on the inputs to the units.

As a simple illustration of the interpretation of network outputs as probabilities, we consider a two-class problem with one input variable in which the class-conditional densities are given by the Gaussian mixture functions shown in Figure 4. A feed-forward network with five hidden units having sigmoidal activation functions, and one output unit having a logistic sigmoid activation function, was trained by minimizing a cross-entropy error using 100 cycles of the BFGS quasi-Newton algorithm (Section 3.3). The resulting network mapping function is shown, along with the true posterior probability calculated using Bayes' theorem, in Figure 5.

For the case of more than two classes, we consider a network with one output for each class so that each output represents the corresponding posterior probability. First of all we choose the target values for network training according to a 1-of-c coding scheme, so that $t_k^n = \delta_{kl}$ for a pattern n from class $\mathcal{C}_l$. We wish to arrange for the probability of observing the set of target values t_k^n, given an input vector $\mathbf{x}^n$, to be given by the corresponding network output so that $p(\mathcal{C}_l|\mathbf{x}) = y_l$. The value of the conditional distribution for this pattern can therefore be written as

$$p(\mathbf{t}^n|\mathbf{x}^n) = \prod_{k=1}^{c} (y_k^n)^{t_k^n}. \tag{35}$$

If we form the likelihood function, and take the negative logarithm as before, we obtain an error function of the form

$$E = -\sum_n \sum_{k=1}^{c} t_k^n \ln y_k^n. \tag{36}$$

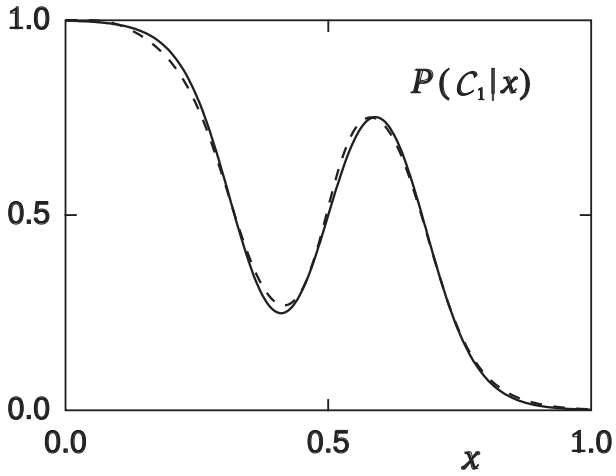

Figure 5. The result of training a multi-layer perceptron on data generated from the density functions in Figure 4. The solid curve shows the output of the trained network as a function of the input variable x, while the dashed curve shows the true posterior probability $P(\mathcal{C}_1|x)$ calculated from the class-conditional densities using Bayes' theorem.

Again we must seek the appropriate output-unit activation function to match this choice of error function. As before, we shall assume that the activations of the hidden units are distributed according to (29). From Bayes' theorem, the posterior probability of class $\mathcal{C}_k$ is given by

$$p(\mathcal{C}_k|\mathbf{z}) = \frac{p(\mathbf{z}|\mathcal{C}_k)P(\mathcal{C}_k)}{\sum_{k'} p(\mathbf{z}|\mathcal{C}_{k'})P(\mathcal{C}_{k'})}. \tag{37}$$

Substituting (29) into (37) and re-arranging we obtain

$$p(\mathcal{C}_k|\mathbf{z}) = y_k = \frac{\exp(a_k)}{\sum_{k'} \exp(a_{k'})} \tag{38}$$

where

$$a_k = \mathbf{w}_k^{\mathrm{T}}\mathbf{z} + w_{k0} \tag{39}$$

and we have defined

$$\mathbf{w}_k = \boldsymbol{\theta}_k \tag{40}$$

$$w_{k0} = A(\boldsymbol{\theta}_k) + \ln P(\mathcal{C}_k). \tag{41}$$

The activation function (38) is called a *softmax* function or *normalized exponential*. It has the properties that $0 \leq y_k \leq 1$ and $\sum_k y_k = 1$ as required for probabilities.

It is easily verified (Bishop, 1995) that the minimization of the error function (36), for an infinite data set and a highly flexible network function, indeed leads to network outputs which represent the posterior probabilities for any input vector $\mathbf{x}$.

Note that the network outputs of the trained network need not be close to 0 or 1 if the class-conditional density functions are overlapping. Heuristic procedures, such as applying extra training using those patterns which fail to generate outputs close to the target values, will be counterproductive, since this alters the distributions and makes it *less* likely that the network will generate the correct Bayesian probabilities!

3.3 Error back-propagation

Using the principle of maximum likelihood, we have formulated the problem of learning in neural networks in terms of the minimization of an error function $E(\mathbf{w})$. This error depends on the vector

$\mathbf{w}$ of weight and bias parameters in the network, and the goal is therefore to find a weight vector $\mathbf{w}^*$ which minimizes E. For models of the form (4) in which the basis functions are fixed, and for an error function given by the sum-of-squares form (25), the error is a quadratic function of the weights. Its minimization then corresponds to the solution of a set of coupled linear equations and can be performed rapidly in fixed time. We have seen, however, that models with fixed basis functions suffer from very poor scaling with input dimensionality. In order to avoid this difficulty we need to consider models with adaptive basis functions. The error function now becomes a highly non-linear function of the weight vector, and its minimization requires sophisticated optimization techniques.

We have considered error functions of the form (25), (28) and (36) which are differentiable functions of the network outputs. Similarly, we have considered network mappings which are differentiable functions of the weights. It therefore follows that the error function itself will be a differentiable function of the weights and so we can use gradient-based methods to find its minima. We now show that there is a computationally efficient procedure, called *back-propagation*, which allows the required derivatives to be evaluated for arbitrary feed-forward network topologies.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$z_j = g(a_j), \qquad a_j = \sum_i w_{ji} z_i \tag{42}$$

where z_i is the activation of a unit, or input, which sends a connection to unit j, and w_{ji} is the weight associated with that connection. The summation runs over all units which send connections to unit j. Biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$. We therefore do not need to deal with biases explicitly. The error functions which we are considering can be written as a sum over patterns of the error for each pattern separately so that $E = \sum_n E^n$. This follows from the assumed independence of the data points under the given distribution. We can therefore consider one pattern at a time, and then find the derivatives of E by summing over patterns.

For each pattern we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (42). This process is often called *forward propagation* since it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E^n with respect to some weight w_{ji}. First we note that E^n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \tag{43}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} \tag{44}$$

where the δ's are often referred to as *errors* for reasons which will become clear shortly. Using (42) we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{45}$$

Substituting (44) and (45) into (43) we then obtain

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_i. \tag{46}$$

Equation (46) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Thus, in order to evaluate the derivatives, we need

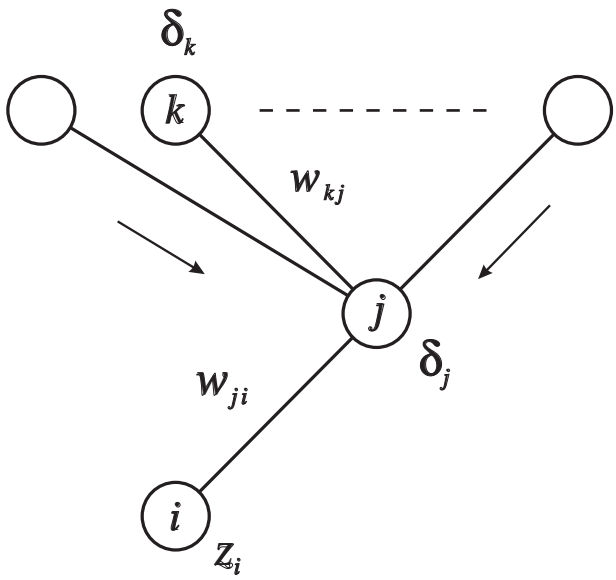

Figure 6. Illustration of the calculation of δ_j for hidden unit j by back-propagation of the δ's from those units k to which unit j sends connections.

only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (46).

For the output units the evaluation of δ_k is straightforward. From the definition (44) we have

$$\delta_k \equiv \frac{\partial E^n}{\partial a_k} = g'(a_k)\frac{\partial E^n}{\partial y_k} \tag{47}$$

where we have used (42) with z_k denoted by y_k. In order to evaluate (47) we substitute appropriate expressions for $g'(a)$ and $\partial E^n/\partial y$. If, for example, we consider the sum-of-squares error function (25) together with a network having linear outputs, as in (7) for instance, we obtain

$$\delta_k = y_k^n - t_k^n \tag{48}$$

and so δ_k represents the error between the actual and the desired values for output k. The same form (48) is also obtained if we consider the cross-entropy error function (28) together with a network with a logistic sigmoid output, or if we consider the error function (36) together with the softmax activation function (38).

To evaluate the δ's for hidden units we again make use of the chain rule for partial derivatives, to give

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{49}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 6. Note that the units labelled k could include other hidden units and/or output units. In writing down (49) we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (44) into (49), and make use of (42), we obtain the following *back-propagation* formula

$$\delta_j = g'(a_j)\sum_k w_{kj}\delta_k \tag{50}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in Figure 6. Since we already

know the values of the δ's for the output units, it follows that by recursively applying (50) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology. Having found the gradient of the error function for this particular pattern, the process of forward and backward propagation is repeated for each pattern in the data set, and the resulting derivatives summed to give the gradient $\nabla E(\mathbf{w})$ of the total error function.

The back-propagation algorithm allows the error function gradient $\nabla E(\mathbf{w})$ to be evaluated efficiently. We now seek a way of using this gradient information to find a weight vector which minimizes the error. This is a standard problem in unconstrained non-linear optimization and has been widely studied, and a number of powerful algorithms have been developed. Such algorithms begin by choosing an initial weight vector $\mathbf{w}^{(0)}$ (which might be selected at random) and then making a series of steps through weight space of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{51}$$

where τ labels the iteration step. The simplest choice for the weight update is given by the gradient descent expression

$$\Delta\mathbf{w}^{(\tau)} = -\eta \left.\nabla E\right|_{\mathbf{w}^{(\tau)}} \tag{52}$$

where the gradient vector ∇E must be re-evaluated at each step. It should be noted that gradient descent is a very inefficient algorithm for highly non-linear problems such as neural network optimization. Numerous *ad hoc* modifications have been proposed to try to improve its efficiency. One of the most common is the addition of a *momentum* term in (52) to give

$$\Delta\mathbf{w}^{(\tau)} = -\eta \left.\nabla E\right|_{\mathbf{w}^{(\tau)}} + \mu\Delta\mathbf{w}^{(\tau-1)} \tag{53}$$

where μ is called the momentum parameter. While this can often lead to improvements in the performance of gradient descent, there are now two arbitrary parameters η and μ whose values must be adjusted to give best performance. Furthermore, the optimal values for these parameters will often vary during the optimization process. In fact much more powerful techniques have been developed for solving non-linear optimization problems (Polak, 1971; Gill *et al.*, 1981; Dennis and Schnabel, 1983; Luenberger, 1984; Fletcher, 1987; Bishop, 1995). These include conjugate gradient methods, quasi-Newton algorithms, and the Levenberg-Marquardt technique.

It should be noted that the term back-propagation is used in the neural computing literature to mean a variety of different things. For instance, the multi-layer perceptron architecture is sometimes called a back-propagation network. The term back-propagation is also used to describe the training of a multi-layer perceptron using gradient descent applied to a sum-of-squares error function. In order to clarify the terminology it is useful to consider the nature of the training process more carefully. Most training algorithms involve an iterative procedure for minimization of an error function, with adjustments to the weights being made in a sequence of steps. At each such step we can distinguish between two distinct stages. In the first stage, the derivatives of the error function with respect to the weights must be evaluated. As we shall see, the important contribution of the back-propagation technique is in providing a computationally efficient method for evaluating such derivatives. Since it is at this stage that errors are propagated backwards through the network, we use the term back-propagation specifically to describe the evaluation of derivatives. In the second stage, the derivatives are then used to compute the adjustments to be made to the weights. The simplest such technique, and the one originally considered by Rumelhart *et al.* (1986), involves gradient descent. It is important to recognize that the two stages are distinct. Thus, the first stage process, namely the propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multi-layer perceptron. It can also be applied to error functions other that the simple sum-of-squares, and to the evaluation of other quantities such as the Hessian matrix whose elements comprise the second derivatives of the error function with respect to the weights (Bishop, 1992). Similarly, the second stage of weight adjustment using the calculated derivatives can be tackled using a variety of optimization schemes (discussed above), many of which are substantially more effective than simple gradient descent.

One of the most important aspects of back-propagation is its computational efficiency. To understand this, let us examine how the number of computer operations required to evaluate the derivatives of the error function scales with the size of the network. A single evaluation of the error function (for a given input pattern) would require $\mathcal{O}(W)$ operations, where W is the total number of weights in the network. For W weights in total there are W such derivatives to evaluate. A direct evaluation of these derivatives individually would therefore require $\mathcal{O}(W^2)$ operations. By comparison, back-propagation allows all of the derivatives to be evaluated using a single forward propagation and a single backward propagation together with the use of (46). Since each of these requires $\mathcal{O}(W)$ steps, the overall computational cost is reduced from $\mathcal{O}(W^2)$ to $\mathcal{O}(W)$. The training of multi-layer perceptron networks, even using back-propagation coupled with efficient optimization algorithms, can be very time consuming, and so this gain in efficiency is crucial.

4 Generalization

The goal of network training is not to learn an exact representation of the training data itself, but rather to build a statistical model of the process which generates the data. This is important if the network is to exhibit good *generalization*, that is, to make good predictions for new inputs.

In order for the network to provide a good representation of the generator of the data it is important that the effective complexity of the model be matched to the data set. This is most easily illustrated by returning to the analogy with polynomial curve fitting introduced in Section 2.1. In this case the model complexity is governed by the order of the polynomial which in turn governs the number of adjustable coefficients. Consider a data set of 11 points generated by sampling the function

$$h(x) = 0.5 + 0.4 \sin(2\pi x) \tag{54}$$

at equal intervals of x and then adding random noise with a Gaussian distribution having standard deviation $\sigma = 0.05$. This reflects a basic property of most data sets of interest in pattern recognition in that the data exhibits an underlying systematic component, represented in this case by the function $h(x)$, but is corrupted with random noise. Figure 7 shows the training data, as well as the function $h(x)$ from (54), together with the result of fitting a linear polynomial, given by (2) with $M = 1$. As can be seen, this polynomial gives a poor representation of $h(x)$, as a consequence of its limited flexibility. We can obtain a better fit by increasing the order of the polynomial, since this increases the number of *degrees of freedom* (i.e. the number of free parameters) in the function, which gives it greater flexibility.

Figure 8 shows the result of fitting a cubic polynomial ($M = 3$) which gives a much better approximation to $h(x)$. If, however, we increase the order of the polynomial too far, then the approximation to the underlying function actually gets worse. Figure 9 shows the result of fitting a 10th-order polynomial ($M = 10$). This is now able to achieve a perfect fit to the training data, since a 10th-order polynomial has 11 free parameters, and there are 11 data points. However, the polynomial has fitted the data by developing some dramatic oscillations and consequently gives a poor representation of $h(x)$. Functions of this kind are said to be *over-fitted* to the data.

In order to determine the generalization performance of the different polynomials, we generate a second independent *test* set, and measure the root-mean-square error E^{RMS} with respect to both training and test sets. Figure 10 shows a plot of E^{RMS} for both the training data set and the test data set, as a function of the order M of the polynomial. We see that the training set error decreases steadily as the order of the polynomial increases. However, the test set error reaches a minimum at $M = 3$, and thereafter increases as the order of the polynomial is increased. The smallest error is achieved by that polynomial ($M = 3$) which most closely matches the function $h(x)$ from which the data was generated.

In the case of neural networks the weights and biases are analogous to the polynomial coefficients. These parameters can be optimized by minimization of an error function defined with respect to a training data set. The model complexity is governed by the number of such parameters and so is determined by the network architecture and in particular by the number of hidden

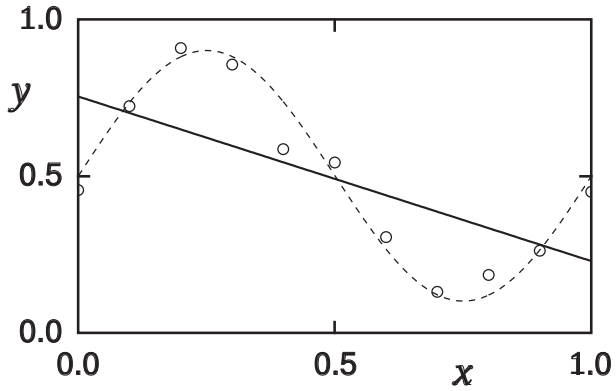

Figure 7. An example of a set of 11 data points obtained by sampling the function $h(x)$, defined by (54), at equal intervals of x and adding random noise. The dashed curve shows the function $h(x)$, while the solid curve shows the rather poor approximation obtained with a linear polynomial, corresponding to $M = 1$ in (2).

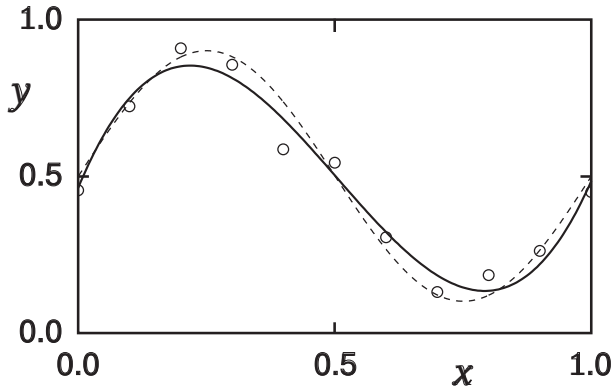

Figure 8. This shows the same data set as in Figure 7, but this time fitted by a cubic ($M = 3$) polynomial, showing the significantly improved approximation to $h(x)$ achieved by this more flexible function.

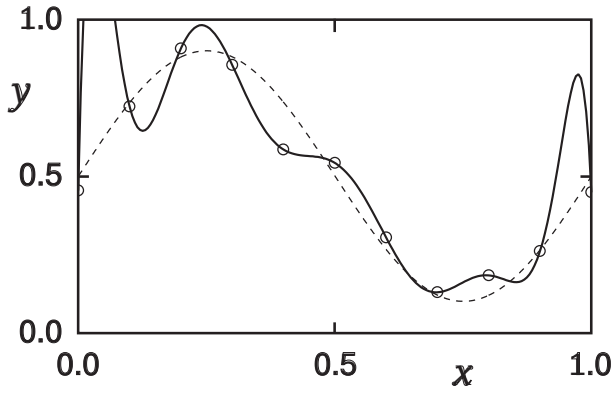

Figure 9. The result of fitting the same data set as in Figure 7 using a 10th-order ($M = 10$) polynomial. This gives a perfect fit to the training data, but at the expense of a function which has large oscillations, and which therefore gives a poorer representation of the generator function $h(x)$ than did the cubic polynomial of Figure 8.

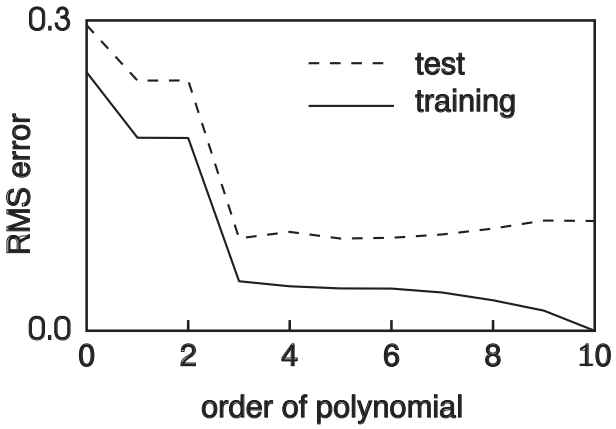

Figure 10. Plots of the RMS error E^{RMS} as a function of the order of the polynomial for both training and test sets, for the example problem considered in the previous three figures. The error with respect to the training set decreases monotonically with M, while the error in making predictions for new data (as measured by the test set) shows a minimum at $M = 3$.

units. We have seen that the complexity cannot be optimized by minimization of training set error since the smallest training error corresponds to an over-fitted model which has poor generalization. Instead, we see that the optimum complexity can be chosen by comparing the performance of a range of trained models using an independent test set. A more elaborate version of this procedure is *cross-validation* (Stone, 1974, 1978; Wahba and Wold, 1975).

Instead of directly varying the number of adaptive parameters in a network, the effective complexity of the model may be controlled through the technique of *regularization*. This involves the use of a model with a relatively large number of parameters, together with the addition of a penalty term Ω to the usual error function E to give a total error function of the form

$$\widetilde{E} = E + \nu\Omega \tag{55}$$

where ν is called a regularization coefficient. The penalty term Ω is chosen so as to encourage smoother network mapping functions since, by analogy with the polynomial results shown in Figures 7–9, we expect that good generalization is achieved when the rapid variations in the mapping associated with over-fitting are smoothed out. There will be an optimum value for ν which can again be found by comparing the performance of models trained using different values of ν on an independent test set. Regularization is usually the preferred choice for model complexity control for a number of reasons: it allows prior knowledge to be incorporated into network training; it has a natural interpretation in the Bayesian framework (discussed in Section 5); and it can be extended to provide more complex forms of regularization involving several different regularization parameters which can be used, for example, to determine the relative importance of different inputs.

5 Discussion

In this chapter we have presented a brief overview of neural networks from the viewpoint of statistical pattern recognition. Due to lack of space, there are many important issues which we have not discussed or have only touched upon. Here we mention two further topics of considerable significance for neural computing.

In practical applications of neural networks, one of the most important factors determining the overall performance of the final system is that of data pre-processing. Since a neural network mapping has universal approximation capabilities, as discussed in Section 2.2, it would in principle be possible to use the original data directly as the input to a network. In practice, however, there is generally considerable advantage in processing the data in various ways before it is used for network training. One important reason why preprocessing can lead to improved performance is that it can offset some of the effects of the 'curse of dimensionality' discussed in Section 2.2 by reducing the number of input variables. Input can be combined in linear or non-linear ways to give a smaller number of new inputs which are then presented to the network. This is sometimes called *feature extraction*. Although information is often lost in the process, this can be more than compensated for by the benefits of a lower input dimensionality. Another significant aspect of pre-processing is that it allows the use of *prior knowledge*, in other words information which is relevant to the solution of a problem which is additional to that contained in the training data. A simple example would be the prior knowledge that the classification of a handwritten digit should not depend on the location of the digit within the input image. By extracting features which are independent of position, this translation invariance can be incorporated into the network structure, and this will generally give substantially improved performance compared with using the original image directly as the input to the network. Another use for preprocessing is to clean up deficiencies in the data. For example, real data sets often suffer from the problem of missing values in many of the patterns, and these must be accounted for before network training can proceed.

The discussion of learning in neural networks given above was based on the principle of maximum likelihood, which itself stems from the *frequentist* school of statistics. A more fundamental, and potentially more powerful, approach is given by the *Bayesian* viewpoint (Jaynes, 1986). Instead of describing a trained network by a single weight vector $\mathbf{w}^*$, the Bayesian approach expresses

our uncertainty in the values of the weights through a probability distribution $p(\mathbf{w})$. The effect of observing the training data is to cause this distribution to become much more concentrated in particular regions of weight space, reflecting the fact that some weight vectors are more consistent with the data than others. Predictions for new data points require the evaluation of integrals over weight space, weighted by the distribution $p(\mathbf{w})$. The maximum likelihood approach considered in Section 3 then represents a particular approximation in which we consider only the most probable weight vector, corresponding to a peak in the distribution. Aside from offering a more fundamental view of learning in neural networks, the Bayesian approach allows error bars to be assigned to network predictions, and regularization arises in a natural way in the Bayesian setting. Furthermore, a Bayesian treatment allows the model complexity (as determined by regularization coefficients for instance) to be treated without the need for independent data as in cross-validation.

Although the Bayesian approach is very appealing, a full implementation is intractable for neural networks. Two principal approximation schemes have therefore been considered. In the first of these (MacKay, 1992a, 1992b, 1992c) the distribution over weights is approximated by a Gaussian centred on the most probable weight vector. Integrations over weight space can then be performed analytically, and this leads to a practical scheme which involves relatively small modifications to conventional algorithms. An alternative approach to the Bayesian treatment of neural networks is to use Monte Carlo techniques (Neal, 1994) to perform the required integrations numerically without making analytical approximations. Again, this leads to a practical scheme which has been applied to some real-world problems.

An interesting aspect of the Bayesian viewpoint is that it is not, in principle, necessary to limit network complexity (Neal, 1994), and that over-fitting should not arise if the Bayesian approach is implemented correctly.

A more comprehensive discussion of these and other topics can be found in Bishop (1995).

References

Anderson, J. A. and E. Rosenfeld (Eds.) (1988). *Neurocomputing: Foundations of Research*. Cambridge, MA: MIT Press.

Baum, E. B. and F. Wilczek (1988). Supervised learning of probability distributions by neural networks. In D. Z. Anderson (Ed.), *Neural Information Processing Systems*, pp. 52–61. New York: American Institute of Physics.

Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. New Jersey: Princeton University Press.

Bishop, C. M. (1992). Exact calculation of the Hessian matrix for the multilayer perceptron. *Neural Computation* **4** (4), 494–501.

Bishop, C. M. (1994). Mixture density networks. Technical Report NCRG/94/001, Neural Computing Research Group, Aston University, Birmingham, UK.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Broomhead, D. S. and D. Lowe (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems* **2**, 321–355.

Cotter, N. E. (1990). The Stone-Weierstrass theorem and its application to neural networks. *IEEE Transactions on Neural Networks* **1** (4), 290–295.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* **2**, 304–314.

Dennis, J. E. and R. B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.

Devijver, P. A. and J. Kittler (1982). *Pattern Recognition: A Statistical Approach*. Englewood Cliffs, NJ: Prentice-Hall.

Duda, R. O. and P. E. Hart (1973). *Pattern Classification and Scene Analysis*. New York: John Wiley.

Fletcher, R. (1987). *Practical Methods of Optimization* (Second ed.). New York: John Wiley.

Friedman, J. H. and W. Stuetzle (1981). Projection pursuit regression. *Journal of the American Statistical Association* **76** (376), 817–823.

Fukunaga, K. (1990). *Introduction to Statistical Pattern Recognition* (Second ed.). San Diego: Academic Press.

Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks* **2** (3), 183–192.

Gill, P. E., W. Murray, and M. H. Wright (1981). *Practical Optimization*. London: Academic Press.

Hampshire, J. B. and B. Pearlmutter (1990). Equivalence proofs for multi-layer perceptron classifiers and the Bayesian discriminant function. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton (Eds.), *Proceedings of the 1990 Connectionist Models Summer School*, pp. 159–172. San Mateo, CA: Morgan Kaufmann.

Hand, D. J. (1981). *Discrimination and Classification*. New York: John Wiley.

Hecht-Nielsen, R. (1989). Theory of the back-propagation neural network. In *Proceedings of the International Joint Conference on Neural Networks*, Volume 1, pp. 593–605. San Diego, CA: IEEE.

Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence* **40**, 185–234.

Hopfield, J. J. (1987). Learning algorithms and probability distributions in feed-forward and feed-back networks. *Proceedings of the National Academy of Sciences* **84**, 8429–8433.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4** (2), 251–257.

Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* **2** (5), 359–366.

Hornik, K., M. Stinchcombe, and H. White (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks* **3** (5), 551–560.

Huber, P. J. (1985). Projection pursuit. *Annals of Statistics* **13** (2), 435–475.

Ito, Y. (1991). Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. *Neural Networks* **4** (3), 385–394.

Jaynes, E. T. (1986). Bayesian methods: general background. In J. H. Justice (Ed.), *Maximum Entropy and Bayesian Methods in Applied Statistics*, pp. 1–25. Cambridge University Press.

Kreinovich, V. Y. (1991). Arbitrary nonlinearity is sufficient to represent all functions by neural networks: a theorem. *Neural Networks* **4** (3), 381–383.

Le Cun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation* **1** (4), 541–551.

Luenberger, D. G. (1984). *Linear and Nonlinear Programming* (Second ed.). Reading, MA: Addison-Wesley.

MacKay, D. J. C. (1992a). Bayesian interpolation. *Neural Computation* **4** (3), 415–447.

MacKay, D. J. C. (1992b). The evidence framework applied to classification networks. *Neural Computation* **4** (5), 720–736.

MacKay, D. J. C. (1992c). A practical Bayesian framework for back-propagation networks. *Neural Computation* **4** (3), 448–472.

Moody, J. and C. J. Darken (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation* **1** (2), 281–294.

Neal, R. M. (1994). *Bayesian Learning for Neural Networks*. Ph.D. thesis, University of Toronto, Canada.

Polak, E. (1971). *Computational Methods in Optimization: A Unified Approach*. New York: Academic Press.

Rumelhart, D. E., R. Durbin, R. Golden, and Y. Chauvin (1995). Backpropagation: the basic theory. In Y. Chauvin and D. E. Rumelhart (Eds.), *Backpropagation: Theory, Architectures, and Applications*, pp. 1–34. Hillsdale, NJ: Lawrence Erlbaum.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations, pp. 318–362. Cambridge, MA: MIT Press. Reprinted in Anderson and Rosenfeld (1988).

Solla, S. A., E. Levin, and M. Fleisher (1988). Accelerated learning in layered neural networks. *Complex Systems* **2**, 625–640.

Stinchecombe, M. and H. White (1989). Universal approximation using feed-forward networks with non-sigmoid hidden layer activation functions. In *Proceedings of the International Joint Conference on Neural Networks*, Volume 1, pp. 613–618. San Diego: IEEE.

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society, B* **36** (1), 111–147.

Stone, M. (1978). Cross-validation: A review. *Math. Operationsforsch. Statist. Ser. Statistics* **9** (1), 127–139.

Wahba, G. and S. Wold (1975). A completely automatic French curve: fitting spline functions by cross-validation. *Communications in Statistics, Series A* **4** (1), 1–17.