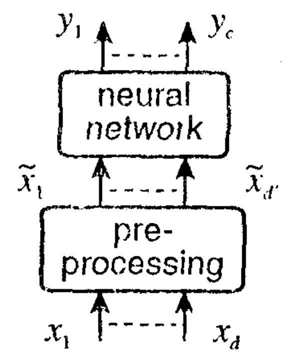

Figure 1.4. The majority of neural network applications require the original input variables $x_1, \ldots, x_d$ to be transformed by some form of pre-processing to give a new set of variables $\widetilde{x}_1, \ldots, \widetilde{x}_{d'}$. These are then treated as the inputs to the neural network, whose outputs are denoted by $y_1, \ldots, y_c$.

1.4 The curse of dimensionality

There is another important reason why pre-processing can have a profound effect on the performance of a pattern recognition system. To see this let us return again to the character recognition problem, where we saw that increasing the number of features from 1 to 2 could lead to an improvement in performance. This suggests that we might use an ever larger number of such features, or even dispense with feature extraction altogether and simply use all 65 536 pixel values directly as inputs to our neural network. In practice, however, we often find that, beyond a certain point, adding new features can actually lead to a *reduction* in the performance of the classification system. In order to understand this important effect, consider the following very simple technique (not recommended in practice) for modelling non-linear mappings from a set of input variables x_i to an output variable y on the basis of a set of training data.

We begin by dividing each of the input variables into a number of intervals, so that the value of a variable can be specified approximately by saying in which interval it lies. This leads to a division of the whole input space into a large number of boxes or cells as indicated in Figure 1.5. Each of the training examples corresponds to a point in one of the cells, and carries an associated value of the output variable y. If we are given a new point in the input space, we can determine a corresponding value for y by finding which cell the point falls in, and then returning the average value of y for all of the training points which lie in that cell. By increasing the number of divisions along each axis we could increase the precision with which the input variables can be specified. There is, however, a major problem. If each input variable is divided into M divisions, then the total number of cells is M^d and this grows *exponentially* with the dimensionality of the input space. Since each cell must contain at least one data point, this implies that the quantity of training data needed to specify the mapping also grows exponentially. This phenomenon has been termed the *curse of dimensionality*

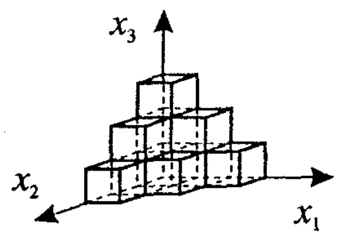

Figure 1.5. One way to specify a mapping from a d-dimensional space $x_1, \ldots, x_d$ to an output variable y is to divide the input space into a number of cells, as indicated here for the case of $d = 3$, and to specify the value of y for each of the cells. The major problem with this approach is that the number of cells, and hence the number of example data points required, grows exponentially with d, a phenomenon known as the 'curse of dimensionality'.

(Bellman, 1961). If we are forced to work with a limited quantity of data, as we are in practice, then increasing the dimensionality of the space rapidly leads to the point where the data is very sparse, in which case it provides a very poor representation of the mapping.

Of course, the technique of dividing up the input space into cells is a particularly inefficient way to represent a multivariate non-linear function. In subsequent chapters we shall consider other approaches to this problem, based on feed-forward neural networks, which are much less susceptible to the curse of dimensionality. These techniques are able to exploit two important properties of real data. First, the input variables are generally correlated in some way, so that the data points do not fill out the entire input space but tend to be restricted to a sub-space of lower dimensionality. This leads to the concept of *intrinsic dimensionality* which is discussed further in Section 8.6.1. Second, for most mappings of practical interest, the value of the output variables will not change arbitrarily from one region of input space to another, but will typically vary smoothly as a function of the input variables. Thus, it is possible to infer the values of the output variables at intermediate points, where no data is available, by a process similar to interpolation.

Although the effects of dimensionality are generally not as severe as the example of Figure 1.5 might suggest, it remains true that, in many problems, reducing the number of input variables can sometimes lead to improved performance for a given data set, even though information is being discarded. The fixed quantity of data is better able to specify the mapping in the lower-dimensional space, and this more than compensates for the loss of information. In our simple character recognition problem we could have considered all 65 536 pixel values as inputs to our non-linear model. Such an approach, however, would be expected to give extremely poor results as a consequence of the effects of dimensionality coupled with a limited size of data set. As we shall discuss in Chapter 8, one of the important roles of pre-processing in many applications is to reduce the dimensionality of the data before using it to train a neural network or other pattern recognition system.

1.5 Polynomial curve fitting

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

$$y(x) = w_0 + w_1 x + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j. \tag{1.2}$$

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 $\mathbf{w}$. The polynomial can then be written as a functional mapping in the form $y = y(x; \mathbf{w})$ as was done for more general non-linear mappings in (1.1).

We shall label the data with the index $n = 1, \ldots, N$, so that each data point consists of a value of x, denoted by x^n, and a corresponding desired value for the output y, which we shall denote by t^n. These desired outputs are called *target* values in the neural network context. In order to find suitable values for the coefficients in the polynomial, it is convenient to consider the error between the desired output t^n, for a particular input x^n, and the corresponding value predicted by the polynomial function given by $y(x^n; \mathbf{w})$. Standard curve-fitting procedures involve minimizing the square of this error, summed over all data points, given by

$$E = \frac{1}{2} \sum_{n=1}^{N} \{y(x^n; \mathbf{w}) - t^n\}^2. \tag{1.3}$$

We can regard E as being a function of $\mathbf{w}$, and so the polynomial can be fitted to the data by choosing a value for $\mathbf{w}$, which we denote by $\mathbf{w}^*$, which minimizes E. Note that the polynomial (1.2) is a linear function of the parameters $\mathbf{w}$ and so (1.3) 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 (Exercise 1.5). Functions which depend linearly on the adaptive parameters are called linear models, even though they may be non-linear functions of the original input variables. Many concepts which arise in the study of such models are also of direct relevance to the more complex non-linear neural networks considered in Chapters 4 and 5. We therefore present an extended discussion of linear models (in the guise of 'single-layer networks') in Chapter 3.