Neural networks for spatial data analysis

Chapter · January 2009

CITATIONS
8

READS
789

1 author:

Manfred M. Fischer
Wirtschaftsuniversität Wien
352 PUBLICATIONS **5,542** CITATIONS

SEE PROFILE

20

Neural Networks for Spatial Data Analysis

Manfred M. Fischer

20.1. INTRODUCTION

The term 'neural network' has its origins in attempts to find mathematical representations of information processing in the study of natural neural systems (McCulloch and Pitts, 1943; Widrow and Hoff, 1960; Rosenblatt, 1962). Indeed, the term has been used very broadly to include a wide range of different model structures, many of which have been the subject of exaggerated claims to mimic neurobiological reality.[1] As rich as neural networks are, they still ignore a host of biologically relevant features. From the perspective of applications in spatial data analysis, however, neurobiological realism is not necessary. In contrast, it would impose entirely unnecessary constraints. Thus, the focus in this chapter is on neural networks as efficient nonlinear models for spatial data analysis. We can not do justice to the entire spectrum of such models. Instead, attention is limited to a particular class of neural network that have proven to be of great practical importance, the class of *feedforward neural networks*.[2]

The attractiveness of such networks is due to two features. *First*, they provide a very flexible framework to approximate arbitrary nonlinear mappings from a set of input variables to a set of output variables where the form of the mapping is governed by a number of adjustable parameters, called weights. *Second*, they are devices for nonparametric statistical inference. No particular structure or parametric form is assumed *a priori*. This is particularly useful in the case of problems where solutions require knowledge that is difficult to specify *a priori*, but for which there are sufficient observations.

The objective of this chapter is to provide an entry point and appropriate background, for those spatial analysts wishing to engage in

[19:20 20/8/2008 5187-Fotheringham-Ch20.tex] Paper: a4 Job No: 5187 Fotheringham: Spatial Analysis (Handbook) Page: 340 340–360

the field of neural networks, required to fully realize its potential. The chapter is organized as follows. In section 20.2 we begin by introducing the functional form of feedforward neural network models, including the specific parameterization of the nonlinear transfer functions. Section 20.3 proceeds to discuss the problem of determining the network parameters within a framework that involves the solution of a nonlinear optimization problem. Because there is no hope of finding an analytical solution to this optimization problem, section 20.4 reviews some of the most important iterative search procedures that utilize gradient information for solving the problem. This requires the evaluation of derivatives of the objective function – known as *error function* in the machine learning literature – with respect to the network parameters, and section 20.5 shows how these can be obtained computationally efficient using the technique of *error backpropagation*. The section that follows addresses the issue of network complexity and briefly discusses some techniques (in particular *regularization* and *early stopping*) to determine the number of hidden units. This problem is shown to essentially consist of optimizing the complexity of the network model (complexity in terms of free parameters) in order to achieve the best *generalization performance*. Section 20.7 then moves attention to the issue of how to appropriately test the generalization performance of a neural network. Some conclusions and an outlook for the future are given in the final section.

The bibliography that is included intends to provide useful pointers to the literature rather than a complete record of the whole field of neural networks. The readers should recognize that there are several wide ranging text books with introductory character, of which Hertz *et al*. (1991), Ripley (1996) and Bishop (2006) appear to be most suitable for a spatial analysis audience. Readers interested in spatial interaction or flow data analysis are referred to a paper by Fischer and Reismann (2002b) to find a useful methodology for neural spatial interaction modelling.

20.2. FEEDFORWARD NEURAL NETWORKS

Feedforward neural networks consist of nodes (also known as processing units or simply units) that are organized in layers. Figure 20.1 shows a schematic diagram of a typical feedforward neural network containing a single intermediate layer of processing units separating input from output units. Intermediate layers of this sort are often called *hidden* layers to distinguish them from the input and output layers. In this network there are N input nodes representing input variables $x_1, \ldots, x_N$; H hidden units representing hidden variables $z_1, \ldots, z_H$; and K output nodes representing output variables $y_1, \ldots, y_K$. Weight parameters are represented by links between the nodes. The bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. Observe the feedforward structure where the inputs are connected only to units in the hidden layer, and the outputs of this layer are connected only to units in the output layer.

The term *architecture* or topology of a network refers to the topological arrangement of the nodes. We call the network architecture shown in Figure 20.1 a single hidden layer network or a two layer rather than a three layer network because it is the number of layers of adaptive weights that is important for determining the network properties. This architecture is most widely used in practice.[3] Kurková (1992) has shown that one hidden layer is sufficient to approximate any continuous function uniformly on a compact input domain. But note that it may be more parsimonious to use fewer hidden units connected in two or more hidden layers.

Any network diagram can be converted into its corresponding mapping function,

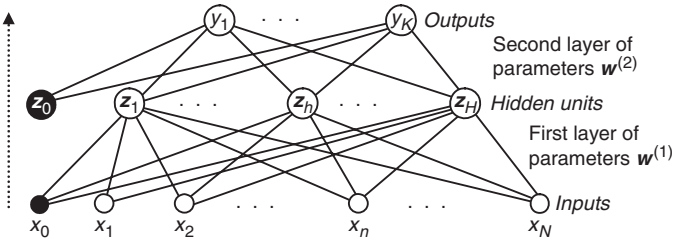

Figure 20.1 Network diagram for the single hidden layer neural network corresponding to equation (20.6). The input, hidden and output variables are represented by nodes, and the weight parameters by links between the nodes, where the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. The arrow denotes the direction of information flow through the network during forward propagation

provided that the diagram is feedforward as in Figure 20.1 so that it does not contain closed directed cycles.[4] This guarantees that the network output y_k $(k = 1, \ldots, K)$ can be described by a series of functional transformations as follows. First, we form a linear combination[5] of the N input variables $x_1, \ldots, x_N$ to get the input, say net_h, that hidden unit h receives:

$$net_h = \sum_{n=1}^{N} w_{hn}^{(1)} x_n + w_{h0}^{(1)} \qquad (20.1)$$

for $h = 1, \ldots, H$. The superscript (1) indicates that the corresponding parameters are in the first layer of the network. The parameters $w_{hn}^{(1)}$ represent connection weights going from input $n\,(n = 1, \ldots, N)$ to hidden unit $h\,(h = 1, \ldots, H)$, and $w_{h0}^{(1)}$ biases.[6] These quantities are known as activations in the field of neural networks. Each of them is then transformed using a differentiable continuous nonlinear or *activation* (transfer) function[7] φ to give to give the output:

$$z_h = \varphi\,(net_h) \qquad (20.2)$$

for $h = 1, \ldots, H$. These quantities are again linearly combined to generate the input, called net_k, that output unit $k\,(k = 1, \ldots, K)$ receives:

$$net_k = \sum_{h=1}^{H} w_{kh}^{(2)} z_h + w_{k0}^{(2)}. \qquad (20.3)$$

The parameters $w_{kh}^{(2)}$ represent the connection weights from hidden unit $h\,(h = 1, \ldots, H)$ to output unit $k\,(k = 1, \ldots, K)$, and the $w_{k0}^{(2)}$ are bias parameters. Finally, the net_k are transformed to produce a set of network outputs $y_k (k = 1, \ldots, K)$:

$$y_k = \psi_k(net_k) \qquad (20.4)$$

where ψ_k denotes a real valued activation function of output unit k.

Information processing in such networks is, thus, straightforward. The input units just provide a 'fan-out' and distribute the input to the hidden units. These units sum their inputs, add a constant (the bias) and take a fixed transfer function φ_h of the result. The output units are of the same form, but with output activation function ψ_k. Network output y_k

can then be expressed in terms of an output function $g_k(\boldsymbol{x}, \boldsymbol{w})$ as:

$$y_k = g_k(\boldsymbol{x}, \boldsymbol{w}) = \psi_k \left(\sum_{h=1}^{H} w_{kh}^{(2)} \varphi_h \times \left(\sum_{n=0}^{N} w_{hn}^{(1)} x_n + w_{h0}^{(1)} \right) + w_{k0}^{(2)} \right) \quad (20.5)$$

where $\boldsymbol{x} = (x_1, \ldots, x_N)$ and $\boldsymbol{w}$ represents a vector of all the weights and bias terms. Note that the bias terms $w_{h0}^{(1)}$ $(h = 1, \ldots, H)$ and $w_{k0}^{(2)}$ $(k = 1, \ldots, K)$ in equation (20.5) can be absorbed[8] into the set of weight parameters by defining additional input and hidden unit variables, x_0 and z_0, whose values are clamped at one so that $x_0 = 1$ and $z_0 = 1$. Then the network function (20.5) becomes

$$y_k = g_k(\boldsymbol{x}, \boldsymbol{w}) = \psi_k \left(\sum_{h=1}^{H} w_{kh}^{(2)} \varphi_h \left(\sum_{n=0}^{N} w_{hn}^{(1)} x_n \right) \right). \quad (20.6)$$

Neural networks of type (20.6) are rather general. They can be seen as a flexible way to parameterize a fairly general nonlinear function from some N-dimensional input space $\boldsymbol{X}$ to some K-dimensional output space $\boldsymbol{Y}$.

Several authors including Cybenko (1989), Funahashi (1989), Hornik *et al.* (1989) and many others have shown that such neural network models, with more or less general types of activation functions φ and ψ, have universal approximation capabilities. They can uniformly approximate any continuous function f on a compact input domain to arbitrary accuracy, provided the network has a sufficiently large number of hidden units. This approximation result, however, is non-constructive and provides no guide to how many hidden units might be needed for a practical problem at hand.

This result holds for a wide range of hidden and output layer activation functions. The functions can be any nonlinearity as long as they are continuous and differentiable. The hidden unit activation functions $\varphi_h(.)$ are typically sigmoid, and almost always taken to be logistic sigmoid[9] so that:

$$\varphi_h(net_k) = \frac{1}{1 + \exp(-net_h)} \quad (20.7)$$

whose outputs lie in the range (0, 1), while the choice of the activation function $\psi_k(.)$ of the output units is generally determined by the nature of data and the assumed distribution of the target variables. Section 20.3 will show that different activation functions should be chosen for different types of problems. For standard regression problems the identity function appears to be an appropriate choice so that $y_k = net_k$. For multiple binary classification each output unit activation should be transformed using a logistic sigmoid function, while the standard multi-class classification problem in which each input is assigned to one of K mutually exclusive classes gives rise to the softmax activation function (Bridle, 1989):

AQ: Not in ref.list.Please check

$$y_k = \psi_k(net_k) = \frac{\exp(net_k)}{\sum_{c=1}^{K} \exp(net_c)} \quad (20.8)$$

where $0 \leq y_k \leq 1$ and $\sum_{k=1}^{K} y_k = 1$.

A neural network with a single logistic output unit can be seen as a nonlinear extension of logistic regression. With many logistic units, it corresponds to linked logistic regressions of each class versus the others. If the transfer functions of the output units in a network are taken to be linear, we have a standard linear model augmented by nonlinear terms. Given the popularity of

linear models in spatial analysis, this form is particularly appealing, as it suggests that neural network models can be viewed as extensions of – rather than as alternatives to – the familiar models. The hidden unit activations can then be viewed as latent variables whose inclusion enriches the linear model.

20.3. NETWORK TRAINING

So far, we have considered neural networks as a general class of parametric nonlinear functions from a vector $\boldsymbol{x}$ of input variables $x_1, \ldots, x_N$ to a vector $\boldsymbol{y}$ of output variables $y_1, \ldots, y_K$. The process of determining the network parameters is called network training or network learning. The problem of determining the network parameters can be viewed from different perspectives. We view it as an unconstrained nonlinear function optimization problem,[10] the solution of which requires the minimization of some (continuous and differentiable) error function.

This error function, say E, is defined in term of deviations of the network outputs $\boldsymbol{y} = (y_1, \ldots, y_K)$ from corresponding desired (target) outputs $\boldsymbol{t} = (t_1, \ldots, t_K)$, and expressed as a function of the weight vector $\boldsymbol{w}$ representing the free parameters (connection weights and bias terms) of the network. The goal of training is then to minimize the error function so that:

$$\min_{\boldsymbol{w} \in \boldsymbol{W}} E(\boldsymbol{w}) \tag{20.9}$$

where $\boldsymbol{W}$ is a weight space appropriate to the network architecture. The smallest value of $E(\boldsymbol{w})$ will occur at a point such that the gradient of the error function vanishes $\nabla E(\boldsymbol{w}) = 0$, where $\nabla E(\boldsymbol{w})$ denotes the gradient (the vector containing the partial derivatives) of $E(\boldsymbol{w})$ with respect to $\boldsymbol{w}$. A single hidden layer network of the kind shown in Figure 20.1, with H hidden units, generally has many points at which the gradient vanishes. The point $\boldsymbol{w}'$ is called a *global* minimum for $E(\boldsymbol{w})$ if $E(\boldsymbol{w}') \leq E(\boldsymbol{w})$ for all $\boldsymbol{w} \in \boldsymbol{W}$. Other minima are called *local* minima, and each corresponds to a different set of parameters. For a successful application of neural networks, however, it may not be necessary to find the global minimum, and in general it will not be known whether the minimum found is the global one or not. But it may be necessary to compare several minima in order to find a sufficiently good solution of the problem under scrutiny.

Training is performed using a training set $S_p = \{(\boldsymbol{x}^p, \boldsymbol{t}^p) : p = 1, \ldots, P\}$, consisting of P ordered pairs of vectors. x^p denotes an N-dimensional input vector and t^p the associated K-dimensional desired output (target) vector. The choice of a suitable error function depends on the problem to be performed. We follow Bishop (1995: chapter 6) to provide a maximum likelihood motivation for the choice, and start by considering regression problems. If we assume that the K target variables are independent conditional on $\boldsymbol{x}$ and $\boldsymbol{w}$ with shared noise precision α, then the conditional distribution of the target values is given by a Gaussian:

$$p(\boldsymbol{t}|\boldsymbol{x}, \boldsymbol{w}) = N(\boldsymbol{t}|\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{w}), \alpha^{-1}\boldsymbol{I}), \tag{20.10}$$

where α is the precision (inverse variance) of the Gaussian noise. For the conditional distribution given by equation (20.10), it is sufficient to take the output unit transfer function ψ to be the identity. Given that $\boldsymbol{t} = (\boldsymbol{t}_1, \ldots, \boldsymbol{t}_P)$, are independent, identically distributed observations, we can construct the corresponding likelihood function:

$$p(\boldsymbol{t}|\boldsymbol{x}, \boldsymbol{w}, \alpha) = \prod_{p=1}^{P} p(\boldsymbol{t}_p|\boldsymbol{x}^p, \boldsymbol{w}, \alpha). \tag{20.11}$$

Maximizing the likelihood function is equivalent to minimizing the *sum-of-squares*

function given by:

$$E(\boldsymbol{w}) = \tfrac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{K} \left\| g_k(\boldsymbol{x}^p, \boldsymbol{w}) - t_k^p \right\|^2. \tag{20.12}$$

The value of $\boldsymbol{w}$ found by solving equation (20.9) will be denoted $\boldsymbol{w}_{ML}$ because it corresponds to the maximum likelihood estimation. Having formed $\boldsymbol{w}_{ML}$, the noise precision is then provided by:

$$\frac{1}{\alpha_{ML}} = \tfrac{1}{PK} \sum_{p=1}^{P} \left\| \boldsymbol{g}(\boldsymbol{x}^p, \boldsymbol{w}_{ML}) - \boldsymbol{t}^p \right\|^2. \tag{20.13}$$

The assumption of independence can be dropped at the expense of a slightly more complex optimization problem. Note that in practice the nonlinearity of the network function $\boldsymbol{g}(\boldsymbol{x}^p, \boldsymbol{w})$ causes the error $E(\boldsymbol{w})$ to be convex, and so in practice local maxima of the likelihood may be found, which correspond to local minima of the error function.

There is a natural pairing of the error function given by the negative log likelihood and the output unit transfer function. In the regression case we can view the network as having a transfer function ψ that is the identity, so that $y_k = net_k$. The corresponding sum-of-squares error function then has the characteristic:

$$\frac{\partial E}{\partial net_k} = (y_k - t_k). \tag{20.14}$$

This property will be used when discussing the technique of error backpropagation in section 20.5.

Now let us consider the case of binary classification where we have a single target variable t such that $t = 1$ denotes class C_1 and $t = 0$ class C_2. We consider a network with a single output whose transfer function is a logistic sigmoid so that $0 \leq g(\boldsymbol{x}, \boldsymbol{w}) \leq 1$, and we can interpret $g(\boldsymbol{x}, \boldsymbol{w})$ as the conditional probability $p(C_1, \boldsymbol{x})$, with $p(C_2, \boldsymbol{x})$ given by $1 - g(\boldsymbol{x}, \boldsymbol{w})$. The conditional probability of targets given inputs is then a Bernoulli distribution of the form:

$$p(t \,|\, \boldsymbol{x}, \boldsymbol{w}) = g(\boldsymbol{x}, \boldsymbol{w})^t \{1 - g(\boldsymbol{x}, \boldsymbol{w})\}^{1-t}. \tag{20.15}$$

If we have a training set of independent observations, then the error function, given by the negative log likelihood, is the cross-entropy error function of the form:

$$E(\boldsymbol{w}) = -\sum_{p=1}^{p} \{t^p \ln y^p + (1 - t^p) \ln(1 - y^p)\} \tag{20.16}$$

where y^p denotes $g(\boldsymbol{x}^p, \boldsymbol{w})$. Note there is no analogue of the noise precision α because the target values are assumed to be correctly labelled.

For classification problems, the targets represent labels defining class membership or – more generally – estimates of the probabilities of class membership. If we have K separate binary classifications to perform, then a neural network with K logistic sigmoid output units is an appropriate choice. In this case a binary class label $t_k^p \in \{0, 1\}$ is associated with each output k. If we assume that the class labels are independent, given the input vector $\boldsymbol{x}^p$, then the conditional distribution is:

$$p(\boldsymbol{t} \,|\, \boldsymbol{x}, \boldsymbol{w}) = \prod_{k=1}^{K} g_k(\boldsymbol{x}, \boldsymbol{w})^{t_k} [1 - g_k(\boldsymbol{x}, \boldsymbol{w})]^{1-t_k}. \tag{20.17}$$

Taking the negative logarithm of the corresponding likelihood function then yields the

multiple-class cross-entropy error function of the form:

$$E(\boldsymbol{w}) = -\sum_{p=1}^{P}\sum_{k=1}^{K}\left\{t_k^p \ln y_k^p + (1-t_k^p)\ln(1-y_k^p)\right\} \quad (20.18)$$

where $y_k^p = g_k(\boldsymbol{x}^p, \boldsymbol{w})$. It is important to note that the derivative of this error function with respect to the activation for a particular output unit k takes the simple form (20.14) as in the regression case.

If we have a standard multiple-class classification problem to solve, where each input is assigned to one of K mutually exclusive classes, then we can use a neural network with K output units each of which has a softmax output activation function. The binary target variables $t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the correct class, and the network outputs are interpreted as $g_k(\boldsymbol{x}^p, \boldsymbol{w}) = p(t_k^p = 1 \mid \boldsymbol{x})$ leading to the error function, called the multiple-class cross-entropy error function (see Fischer and Staufer, 1999):

$$E(\boldsymbol{w}) = -\sum_{p=1}^{P}\sum_{k=1}^{K} t_k^p \ln\left(\frac{g_k(\boldsymbol{x}^p, \boldsymbol{w})}{t_k^p}\right) \quad (20.19)$$

which is non-negative, and equals zero when $g_k(\boldsymbol{x}^p, \boldsymbol{w}) = t_k^p$ for all k and p. Once again, the derivative of this error function with respect to the activation for a particular output unit k takes the familiar form equation (20.14). It is worth noting that in the case of $K = 2$ we can use a network with a single logistic sigmoid output, alternatively to a network with two softmax output activations.

In summary, there is natural pairing of the choice of the output unit transfer function and the choice of the error function, according to the type of the problem that has to be solved. For regression we take linear outputs and a sum-of-squares error, for (multiple independent) binary classifications we use logistic sigmoid outputs with the corresponding cross-entropy error function, and for multi-class classification softmax outputs and the multi-class cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively we can take a network with two softmax outputs (Bishop, 2006: 236).

20.4. PARAMETER OPTIMIZATION

There are many ways to solve the minimization problem (20.9). Closed-form optimization via the calculus of scalar fields rarely admits a direct solution. A relatively new set of interesting techniques that use optimality conditions from calculus are based on evolutionary computation (Goldberg, 1989; Fogel, 1995). But gradient procedures which use the first partial derivatives $\nabla E(\boldsymbol{w})$, so-called first order strategies, are most widely used. Gradient search for solutions gleans its information about derivatives from a sequence of function values. The recursion scheme is based on the formula:[11]

$$\boldsymbol{w}(\tau + 1) = \boldsymbol{w}(\tau) + \eta(\tau)\boldsymbol{d}(\tau) \quad (20.20)$$

where τ denotes the iteration step. Different procedures differ from each other with regard to the choice of step length $\eta(\tau)$ and search direction $\boldsymbol{d}(\tau)$, the former being a scalar called *learning* rate and the latter a vector of unit length.

The simplest approach to using gradient information is to assume $\eta(\tau)$ being constant and to choose the parameter update in equation (20.20) to comprise a small step in the direction of the negative gradient so that:

$$\boldsymbol{d}(\tau) = -\nabla E(\boldsymbol{w}(\tau)). \quad (20.21)$$

After each such update, the gradient is re-evaluated for the new parameter vector $\boldsymbol{w}(\tau + 1)$. Note that the error function is defined with respect to a training set S_P to be processed to evaluate ∇E. One complete presentation of the entire training set during the training process is called an *epoch*. The training process is maintained on an epoch-by-epoch basis until the connection weights and bias terms of the network stabilize and the average error over the entire training set converges to some minimum. It is good practice to randomize the order of presentation of training examples from one epoch to the next. This randomization tends to make the search in the parameter space stochastic over the training cycles, thus avoiding the possibility of limit cycles in the evolution of the weight vectors.

Gradient descent optimization may proceed in one of two ways: pattern mode and batch mode. In the *pattern mode* weight updating is performed after the presentation of each training example. Note that the error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point. Thus:

$$E(\boldsymbol{w}) = \sum_{p=1}^{P} E_p(\boldsymbol{w}) \tag{20.22}$$

where E_p is called the *local error* while E the global error, and pattern mode gradient descent makes an update to the parameter vector based on one training example at a time so that:

$$\boldsymbol{w}(\tau + 1) = \boldsymbol{w}(\tau) - \eta \nabla E_p(\boldsymbol{w}(\tau)). \tag{20.23}$$

Rumelhart *et al*. (1986) have shown that pattern based gradient descent minimizes equation (20.22), if the learning parameter η is sufficiently small. The smaller η, the smaller will be the changes to the weights in the network from one iteration to the next and the smoother will be the trajectory in the parameter space. This improvement, however, is attained at the cost of a slower rate of training. If we make the learning rate parameter η too large so as to speed up the rate of training, the resulting large changes in the parameter weights assume such a form that the network may become unstable.

In the *batch mode* of training, parameter updating is performed after the presentation of all the training examples that constitute an epoch. From an online operational point of view, the pattern mode of training is preferred over the batch mode, because it requires less local storage for each weight connection. Moreover, given that the training patterns are presented to the network in a random manner, the use of pattern-by pattern updating of parameters makes the search in parameter space stochastic in nature[12] which in turn makes it less likely to be trapped in a local minimum. On the other hand, the use of batch mode of training provides a more accurate estimation of the gradient vector ∇E. Finally, the relative effectiveness of the two training modes depends on the problem to be solved (Haykin, 1994: 152 pp).

For batch optimization there are more efficient procedures, such as conjugate gradients and quasi-Newton methods, that are much more robust and much faster than gradient descent (Nocedal and Wright, 1999). Unlike steepest gradient, these algorithms have the characteristic that the error function always decreases at each iteration unless the parameter vector has arrived at a local or global minimum. Conjugate gradient methods achieve this by incorporating an intricate relationship between the direction and gradient vectors. The initial direction vector $\boldsymbol{d}(0)$ is set equal to the negative gradient vector at the initial step $\tau = 0$. Each successive direction vector is then computed as a linear combination of the current gradient vector and the previous direction vector. Thus:

$$\boldsymbol{d}(\tau + 1) = -\nabla E(\boldsymbol{w}(\tau + 1)) + \beta(\tau)\boldsymbol{d}(\tau) \tag{20.24}$$

where $\beta(\tau)$ is a time varying parameter. There are various rules for determining $\beta(\tau)$ in terms of the gradient vectors at time τ and $\tau + 1$ leading to the Fletcher–Reeves and Polak–Ribière variants of conjugate gradient algorithms (see Press *et al*., 1992). The computation of the learning rate parameter $\eta(\tau)$ in the update formula (20.20) involves a line search, the purpose of which is to find a particular value of η for which the error function $E(\boldsymbol{w}(\tau)+\eta\,\boldsymbol{d}(\tau))$ is minimized, given fixed values of $\boldsymbol{w}(\tau)$ and $\boldsymbol{d}(\tau)$.

The application of Newton's method to the training of neural networks is hindered by the requirement of having to calculate the Hessian matrix and its inverse, which can be computationally expensive. The problem is further complicated by the fact that the Hessian matrix $\boldsymbol{H}$ would have to be non-singular for its inverse to be computed. Quasi-Newton methods avoid this problem by building up an approximation to the inverse Hessian over a number of iteration steps. The most commonly variants are the Davidson–Fletcher–Powell and the Broyden–Fletcher–Goldfarb–Shanno procedures (see Press *et al*., 1992).

Quasi-Newton procedures are today the most efficient and sophisticated (batch) optimization algorithms. But they require the evaluation and storage in memory of a dense matrix $\boldsymbol{H}(\tau)$ at each iteration step τ. For larger problems (more than 1,000 weights) the storage of the approximate Hessian can be too demanding. In contrast, the conjugate gradient procedures require much less storage, but an exact determination of the learning rate $\eta(\tau)$ and the parameters $\beta(\tau)$ in each iteration τ, and, thus, approximately twice as many gradient evaluations as the quasi-Newton methods.

When the surface modelled by the error function in its parameter space is extremely rugged and has many local minima, then a local search from a random starting point tends to converge to a local minimum close to the initial point and to a solution worse than the global minimum. In order to seek out good local minima, a good training procedure must thus include both a gradient based optimization algorithm and a technique like random start that enables sampling of the space of minima. Alternatively, stochastic global search procedures might be used. Examples of such procedures include Alopex (see Fischer *et al*., 2003, for an application in the context of spatial interaction data analysis), genetic algorithms (see Fischer and Leung, 1998, for another application in the same context), and simulated annealing. These procedures guarantee convergence to a global solution with high probability, but at the expense of slower convergence.

Finally, it is worth noting that the question whether neural networks can have real-time learning capabilities is still challenging and open. Real-time learning is highly required by time-critical applications, such as for navigation and tracking systems in a GIS-T context, where the data observations are arriving in a continuous stream, and predictions have to be made before all the data seen. Even for offline applications, speed is still a need, and real-time learning algorithms that reduce training time are of considerable value.

20.5. ERROR BACKPROPAGATION

One of the greatest breakthroughs in neural network modelling has been the introduction of the technique of error backpropagation[13] in that it provides a computationally efficient technique to calculate the gradient vector of an error function for a feedforward neural network with respect to the parameters. This technique – sometimes simply termed backprop – uses a local message passing scheme in which information is sent alternately forwards and backwards through the network. Its modern form stems from Rumelhart *et al*. (1986), illustrated for gradient descent optimization applied to the sum-of-squares error function. It is important to recognize, however, that error backpropagation can also be applied to error functions other than just sum-of-squares and to a wide variety of optimization schemes

for weight adjustment other than gradient descent, in pattern or batch mode.

We describe the backpropagation algorithm for a general network of type (20.6) that has a single hidden layer, arbitrary differentiable activation functions with a corresponding local error function $E_p(\boldsymbol{w})$. For each pattern p in the training data set, we shall assume that we have supplied the corresponding input vector $\boldsymbol{x}^p$ to the network and calculated the activations of all the hidden and output units in the network by applying equations (20.1)–(20.4). Recall that each hidden unit h has input net_h^p and output $z_h^p = \varphi_h(net_h^p)$, and each output unit k has input net_k^p and output $y_k^p = \psi_k(net_k^p)$. This process is called forward propagation because it can be seen as a forward flow of information (signals) provided by $\boldsymbol{x}^p$ through the network. For the rest of this section we consider one example and drop the superscript p in order to keep the notation uncluttered.

We evaluate the gradient E_p with respect to a hidden-to-output parameter $w_{kh}^{(2)}$ first, by noting that E_p depends on the weight $w_{kh}^{(2)}$ only via the summed input, net_k, to the output unit k. Thus, we can apply the chain rule for partial derivatives to get:

$$\frac{\partial E_p}{\partial w_{kh}^{(2)}} = \frac{\partial E_p}{\partial net_k}\frac{\partial net_k}{\partial w_{kh}^{(2)}} \tag{20.25}$$

where:

$$\frac{\partial net_k}{\partial w_{kh}^{(2)}} = \frac{\partial}{\partial w_{kh}^{(2)}}\sum_{h=0}^{H} w_{kh}^{(2)} z_h = z_h. \tag{20.26}$$

If we define:

$$\delta_k := \frac{\partial E_p}{\partial net_k} = \psi'(net_k)\frac{\partial E_p}{\partial y_k} \tag{20.27}$$

where the δs are often referred to as *errors*, and substitute equations (20.26) and (20.27) into equation (20.25), we obtain:

$$\frac{\partial E_p}{\partial w_{kh}^{(2)}} = \delta_k z_h. \tag{20.28}$$

This equation tells us that the required partial derivative with respect to $w_{kh}^{(2)}$ is obtained simply by the multiplication of two expressions: the value of δ for unit k at the output end of the connection concerned and the value of z at the input end h of the connection. Thus, in order to evaluate the partial derivatives with respect to the second layer parameters we need only to compute the value of δ_k for each output unit $k = 1, \ldots, K$ in the network, and then apply equation (20.28).

For linear outputs associated with the sum-of-squares error function, for logistic sigmoid outputs associated with the cross-entropy error function and for softmax outputs associated with the multiple-class cross-entropy error function, the δs are given by:

$$\delta_k = y_k - t_k \tag{20.29}$$

while for logistic sigmoid outputs associated with sum-of-squares error function the δs are found as:

$$\delta_k = y_k(1 - y_k)(y_k - t_k). \tag{20.30}$$

For the input-to-hidden connections we must differentiate the chosen error function with respect to the parameters $w_{hn}^{(1)}$, which are more deeply embedded in the error function. Using again the chain rule for partial derivatives, we get:

$$\frac{\partial E_p}{\partial w_{hn}^{(1)}} = \delta_h \frac{\partial net_h}{\partial w_{hn}^{(1)}} = \delta_h x_n \tag{20.31}$$

with:

$$\delta_h := \varphi'_h(net_h) \sum_{k=1}^{K} \delta_k w_{kh}^{(2)} \qquad (20.32)$$

where the use of the prime signifies differentiation with respect to the argument. In the case of logistic hidden units we get the following backpropagation formula:

$$\begin{aligned} \delta_h &= \varphi'_h(net_h) \sum_{k=1}^{K} \delta_k w_{kh}^{(2)} \\ &= \varphi(net_h)(1 - \varphi(net_h)) \sum_{k=1}^{K} \delta_k w_{kh}^{(2)} \\ &= z_h(1 - z_h) \sum_{k=1}^{K} \delta_k w_{kh}^{(2)}. \qquad (20.33) \end{aligned}$$

Since the formula for δ_h contains only terms in a later layer, it is clear that it can be calculated from output to input on the network. Thus, the basic idea behind the technique of error backpropagation is to use a forward pass through the network to calculate the z_h and y_k values by propagating the input vector, followed by a backward pass to calculate δ_k and δ_h, and hence the partial derivatives of the error function. Note that for the presentation of each training example the input pattern is fixed throughout the message passing scheme, encompassing the forward pass followed by the backward pass.

The backpropagation technique can be summarized in the following four steps:

Step 1 Apply an input vector $\boldsymbol{x}^p$ to the network and forward propagate through the network, using equations (20.1)–(20.4), to generate the hidden and output unit activations based on current weight settings.

Step 2 Evaluate the δ_k for all the output units ($k = 1, \ldots, K$) using equation (20.29) or equation (20.30), depending on the problem type to be studied.

Step 3 Backpropagate the deltas, using equation (20.33), to get δ_h for each hidden unit $h (h = 1, \ldots, H)$ in the network.

Step 4 Use equations (20.28) and (20.31) to evaluate the required derivatives.

For batch procedures the gradient of the global error can be obtained by repeating *Step 1* to *Step 4* for each pattern p in the training set, and then summing over all patterns.

20.6. NETWORK COMPLEXITY

So far we have considered neural networks of type (20.6) with *a priori* given numbers of input, hidden and output units. While the number of input and output units in a neural network is basically problem dependent, the number H of hidden units is a free parameter that can be adjusted to provide the best testing performance on independent data, called testing set. But the testing error is not a simple function of H due to the presence of local minima in the error function. The issue of finding a parsimonious model for a real world problem is critical for all models but particularly important for neural networks because the problem of overfitting is more likely to occur.

A neural network model that is too simple (i.e., small H), or too inflexible, will have a large bias and smooth out some of the underlying structure in the data (corresponding to high bias), while one that has too much flexibility in relation to the particular data set will overfit the data and have a large variance. In either case, the performance of the network on new data (i.e., generalization performance) will be poor. This highlights the need to optimize the complexity in the model selection process in order to achieve the best generalization (Bishop, 1995: 332; Fischer, 2000). There are some ways to control the complexity of a neural network, complexity in terms of the number of hidden units or, more precisely, in terms of the independently adjusted

parameters. Practice in spatial data analysis generally adopts a trial and error approach that trains a sequence of neural networks with an increasing number of hidden units and then selects that one which gives the predictive performance on a testing set.[14]

There are, however, other more principled ways to control the complexity of a neural network model in order to avoid overfitting.[15] One approach is that of *regularization*, which involves adding a regularization term $R(\boldsymbol{w})$ to the error function in order to control overfitting, so that the total error function to be minimized takes the form:

$$\tilde{E}_p(\boldsymbol{w}) = E_p(\boldsymbol{w}) + \mu R(\boldsymbol{w}) \qquad (20.34)$$

where μ is a positive real number, the so-called regularization parameter, that controls the relative importance of the data dependent error $E_p(\boldsymbol{w})$ and the regularization term $R(\boldsymbol{w})$, sometimes also called complexity term. This term embodies the *a priori* knowledge about the solution, and therefore depends on the nature of the particular problem to be solved. Note that $\tilde{E}_p(\boldsymbol{w})$ is called the *regularized error function*.

One of the simplest forms of regularizer is defined as the squared norm of the parameter vector $\boldsymbol{w}$ in the network, as given by:

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|^2 \qquad (20.35)$$

AQ: Not in ref.list.Please check

This regularizer[16] is known as a weight decay function that penalizes large weights. Hinton (1987) has found empirically that a regularizer of this form can lead to significant improvements in network generalization.

Sometimes, a more general regularizer is used, for which the regularized error takes the form:

$$E(\boldsymbol{w}) + \mu \|\boldsymbol{w}\|^m \qquad (20.36)$$

where $m = 2$ corresponds to the quadratic regularizer (20.35). The case $m = 1$ is known as the 'lasso' in the statistics literature (Tibshirani, 1996b). It has the property that – if μ is sufficiently large – some of the parameter weights are driven to zero in sequential learning algorithms, leading to a sparse model. As μ is increased, so an increasing number of parameters are driven to zero.

One of the limitations of this regularizer is inconsistency with certain scaling characteristics of network mappings. If one trains a network using original data and one network using data for which the input and/or target variables are linearly transformed, then consistency requires obtaining equivalent networks which differ only by a linear transformation of the weights. Any regularizer should possess this characteristic, otherwise one solution is arbitrary favoured over an equivalent solution. In particular, the weights should be scale invariant (Bishop, 2006: 257–258). A regularized error function that satisfies this property is given by:

$$E(\boldsymbol{w}) + \mu_1 \|\boldsymbol{w}_{q1}\|^m + \mu_2 \|\boldsymbol{w}_{q2}\|^m \qquad (20.37)$$

where $\boldsymbol{w}_{q1}$ denotes the set of the weights in the first layer, that is $w_{11}^{(1)}, \ldots, w_{h1}^{(1)}, \ldots, w_{HN}^{(1)}$, and $\boldsymbol{w}_{q2}$ those in the second layer, that is $w_{11}^{(2)}, \ldots, w_{kh}^{(2)}, \ldots, w_{KH}^{(2)}$. Under linear transformations of the weights, the regularizer will remain unchanged, provided that the parameters μ_1 and μ_2 are suitably rescaled.

The more sophisticated control of complexity that regularization offers over adjusting the number of hidden units by trial and error is evident. Regularization allows complex neural network models to be trained on data sets of limited size without severe overfitting, by limiting the effective network complexity. The problem of determining the appropriate number of hidden units is, thus, shifted to one of determining a suitable value for the regularization parameter(s) during the training process.

The principal alternative to regularization as a way to optimize the model complexity for a given training data set is the procedure of *early stopping*. As we have seen in the previous sections, training of a nonlinear network model corresponds to an iterative reduction of the error function defined with respect to a given training data set. For many of the optimization procedures used for network training (such as conjugate gradient optimization) the error is a nondecreasing function of the iteration steps τ. But the error measured with respect to independent data, called *validation data set*, often shows a decrease first, followed by an increase as the network starts to overfit, as illustrated in Fischer and Gopal (1994) for a spatial interaction data analysis problem. Thus, training can be stopped at the point of smallest error with respect to the validation data, in order to get a network that shows good generalization performance. But, if the validation set is small, it will give a relatively noisy estimate of generalization performance, and it may be necessary to keep aside another data set, the test set, on which the performance of the network model is finally evaluated.

This approach of stopping training before a minimum of the training error has been reached is another way of eliminating the network complexity. It contrasts with regularization because the determination of the number of hidden units does not require convergence of the training process. The training process is used here to perform a directed search of the weight space for a neural network model that does not overfit the data and, thus, shows superior generalization performance. Various theoretical and empirical results have provided strong evidence for the efficiency of early stopping (see, e.g., Weigend *et al*., 1991; Baldi and Chauvin, 1991; Finnoff, 1991). Although many questions remain, a picture is starting to emerge as to the mechanisms responsible for the effectiveness of this procedure. In particular, it has been shown that stopped training has the same sort of regularization effect (i.e., reducing model variance at the cost of bias) that penalty terms provide.

20.7. GENERALIZATION PERFORMANCE

The ability of a neural network to predict correctly new observations that differ from those used for training is known as *generalization* (see, e.g., Moody, 1992). To assess the generalization performance of a neural network model is of crucial importance. The performance on the training set is not a good indicator due to the problem of overfitting. As often in statistics, there is a trade-off between accuracy on the training data and generalization. This is a well-studied dilemma (see, e.g., Bishop, 1995: chapter 9).

The simplest way to assess the generalization performance is the use of a test set. Here, of course, it is assumed that the test data are drawn from the same population used to generate the training data. If the test set is too small, an accurate assessment cannot be obtained. Test set validation becomes practical only if the data sets are very large or new data can be generated cheaply. As the training and test sets are independent samples, an unbiased estimate of the prediction risk is obtained. But the estimate can be highly variable across different data splittings.

One way to overcome this problem is by *cross-validation*. Cross-validation is a sample re-use method for assessing generalization performance. It makes maximally efficient use of the available data. The idea is to divide the available data set into – generally equally sized – D parts, and then to use one part to test the performance of the neural network model trained on the remaining $(D-1)$ parts. The resulting estimator is again unbiased, and we can average the D such estimates. Leave-one-out cross-validation is a special case, in which each observation is tested on the remaining $(P-1)$ observations.

This version evidently requires a large number of computations. Choosing $D = P$ should give the most accurate assessment, as the 'true' size of the training set is most closely mimicked, but also involves the most computation. In addition, cross-validation estimates of performance for large D might be expected to be rather variable. Taking a smaller D can give a larger bias, but smaller variance and mean-square error. This is an argument in favour of smaller D (Ripley, 1996: 70–71). Bootstrap estimates of bias can be used for bias correction (Efron, 1982).

With small samples of data – precisely when structural uncertainty is greatest – cross-validation may not be feasible, because there are too few data values with which to carry out the estimation, validation and testing activities in a stable way. *Bootstrapping* the neural network modelling process – creating bootstrap copies of the available data to generate copies of training, validation and test sets – may be used instead as a general framework for evaluating generalization performance. The idea underlying the bootstrap is appealingly simple. For an introduction see, e.g., Efron (1982), Efron and Tibshirani (1993) or Hastie *et al*. (2001).

Suppose, we are interested in a single hidden layer neural network together with a sum-of-squares error function to solve a regression problem. The standard procedure for estimating and evaluating the neural network is to split the available data set, say $S_P = \{\boldsymbol{z}^p = (\boldsymbol{x}^p, \boldsymbol{t}^p) : p = 1, \ldots, P\}$, into three parts: a training set $S_{P1} = \{\boldsymbol{z}^{p1} = (\boldsymbol{x}^{p1}, \boldsymbol{t}^{p1}) : p1 = 1, \ldots, P1\}$, a validation set $S_{P2} = \{\boldsymbol{z}^{p2} = (\boldsymbol{x}^{p2}, \boldsymbol{t}^{p2}) : p2 = 1, \ldots, P2\}$ and a test set $S_{P3} = \{\boldsymbol{z}^{p3} = (\boldsymbol{x}^{p3}, \boldsymbol{t}^{p3}) : p3 = 1, \ldots, P3\}$, with $P1 + P2 + P3 = P$. The training set serves for parameter estimation such as by means of gradient descent on the sum-of-squares error function. The validation set is used, for example, to determine the stopping point before overfitting occurs, and the test set to evaluate the generalization performance of the model, using some measure of error between a prediction and an observed value, such as the familiar root-mean-square error ρ of the form:

$$\hat{\rho}(\hat{\boldsymbol{w}}) = \frac{\sum_{p3=1}^{P3} \left\| \boldsymbol{g}(\boldsymbol{x}^{p3}, \hat{\boldsymbol{w}}) - \boldsymbol{t}^{p3} \right\|^2}{\sum_{p3=1}^{P3} \left\| \boldsymbol{t}^{p3} - \bar{\boldsymbol{t}} \right\|^2} \tag{20.38}$$

which is a function of $\hat{\boldsymbol{w}}$, obtained by solving the minimization problem (20.9). $\bar{\boldsymbol{t}}$ is defined to be the average test set target vector. Care has to be taken in interpreting the results obtained as accurate estimates of the generalization performance.

Randomness enters into this standard approach to neural network modelling in two ways: in the splitting of the data samples, and in choices about the parameter initialization. This leaves one question wide open. What is the variation of test performance as one varies training, validation, and test sets? This is an important question, since there is not just one 'best' split of the data or obvious choice for the initial weights. Thus, it is useful to vary both the data partitions and parameter initializations to find out more about the distributions of generalization errors. One way is to use a computer intensive bootstrapping approach to evaluate the performance, reliability, and robustness of the neural network model, an approach that combines the purity of splitting the data into three disjoint data sets with the power of a resampling procedure. Implementing this approach involves the following steps (see Fischer and Reismann, 2002a,b, for an application in the context of spatial interaction modelling).

Step 1: Generation of bootstrap training, validation and test sets

Using the sample S_P, we first build a test set by choosing $P3$ patterns randomly,[17] with replacement. The patterns used in this specific test set are then removed from the pool S_P. From the remainder,

we then randomly set aside $P2$ patterns for the bootstrap validation set. They are picked randomly without replacement and removed from the pool. The remaining patterns constitute the training set. This process is repeated B times (typically $20 < B < 200$) to generate $b = 1, \ldots, B$ training data sets of size $P1$, $S_{P1}^{*b} = \{{}^{*b}\boldsymbol{z}^{p1} : p1 = 1, \ldots, P1\}$, called bootstrap training sets; $b = 1, \ldots, B$ validation data sets of size $P2$, $S_{P2}^{*b} = \{{}^{*b}\boldsymbol{z}^{p2} : p2 = 1, \ldots, P2\}$, called bootstrap validation sets; and $b = 1, \ldots, B$ test data sets of size $P3$, $S_{P3}^{*b} = \{{}^{*b}\boldsymbol{z}^{p3} : p3 = 1, \ldots, P3\}$, called bootstrap test sets.

Step 2: Computation of the bootstrap parameter estimates

Each bootstrap training set S_{P1}^{*b} is used to compute a new parameter vector by minimizing:

$$\arg\min \{E({}^{*b}\boldsymbol{w}) : {}^{*b}\boldsymbol{w} \in \boldsymbol{W},\ \boldsymbol{W} \subseteq R^{Q}\} \quad (20.39)$$

where Q is the number of parameters, and $E({}^{*b}\boldsymbol{w})$ the (global) sum-of-squares error for the bth bootstrap training sample. This is given by:

$$E({}^{*b}\boldsymbol{w}) = \tfrac{1}{2} \sum_{p1=1}^{P1} \left\| \boldsymbol{g}({}^{*b}\boldsymbol{x}^{p1},\ {}^{*b}\boldsymbol{w}) - {}^{*b}\boldsymbol{t}^{p1} \right\|^2 \quad (20.40)$$

where the sum runs over the bth bootstrap training set, and $b = 1, \ldots, B$. The corresponding bootstrap validation set is used to determine the stopping point before overfitting occurs and/or to set additional parameters or hyperparameters. This yields B bootstrap parameter estimates ${}^{*b}\hat{\boldsymbol{w}}$(b=1,. . ., B).

Step 3: Estimation of the bootstrap statistic of interest

From S_{P3}^{*b} calculate ${}^{*b}\hat{\rho}\,({}^{*b}\hat{\boldsymbol{w}})$, the bootstrap analogue of $\hat{\rho}\,(\hat{\boldsymbol{w}})$ given by equation (20.38), in the same manner as $\hat{\rho}\,(\hat{\boldsymbol{w}})$ but with resample S_{P3}^{*b} replacing S_{P3} and ${}^{*b}\hat{\boldsymbol{w}}$ replacing $\hat{\boldsymbol{w}}$. This yields a sequence of bootstrap statistics, ${}^{*1}\hat{\rho}, \ldots, {}^{*B}\hat{\rho}$.

Step 4: Estimation of the standard deviation

The statistical accuracy of the performance statistic can then be evaluated by looking at the variability of the statistic between the different bootstrap test sets. Estimate the standard deviation, $\hat{\sigma}$, of ${}^{*}\hat{\rho}$ as approximated by bootstrap:

$$\hat{\sigma}_{P3}^{B} = \left\{ \frac{1}{B-1} \sum_{b=1}^{B} \left[{}^{*b}\hat{\rho}\,({}^{*b}\boldsymbol{w}) - {}^{*}\,\hat{\rho}(.) \right]^2 \right\}^{1/2} \quad (20.41)$$

where:

$${}^{*}\hat{\rho}(.) = \frac{1}{B} \sum_{b=1}^{B} {}^{*b}\hat{\rho}({}^{*b}\boldsymbol{w}). \quad (20.42)$$

The true standard error of $\hat{\rho}$ is a function of the unknown density function F of ρ, that is $\sigma(F)$. With the bootstrapping approach described above one obtains $\hat{F}_{P3}^{*}$ which is supposed to describe closely the empirical probability distribution $\hat{F}_{P3}$, in other words $\hat{\sigma}_{P3}^{B} \approx \sigma(\hat{F}_{P3})$. Asymptotically, this means that as $P3$ tends to infinity, the estimate $\hat{\sigma}_{P3}^{B}$ tends to $\sigma(F)$. For finite sample sizes, however, generally there will be deviations.

Step 5: Bias estimation

The bootstrap scheme can be used to estimate not only the variability of the performance statistic $\hat{\rho}$, but also its bias (Zapranis and Refenes, 1998). Bias can be thought of as a function of the unknown probability density function F of ρ that is $\beta = \beta(F)$. The bootstrap estimate of bias is simply:

$$\hat{\beta}_B = \beta(\hat{F}_{P3}) = E^{*}[\rho(\hat{F}_{P3}^{*}) - \rho(\hat{F}_{P3})] \quad (20.43)$$

where E^{*} indicates expectation with respect to bootstrap sampling and $\hat{F}_{P3}^{*}$

the bootstrap empirical distribution. The bootstrap estimate of bias is:

$$\hat{\beta} = \frac{1}{B} \sum_{b=1}^{B} \left[{}^{*b}\hat{\rho}\,({}^{*b}\boldsymbol{w}) - \hat{\rho}\,(\boldsymbol{w}) \right] \tag{20.44}$$

The bias is removed by subtracting $\hat{\beta}_B$ from the estimated $\hat{\rho}$.

20.8. SUMMARY AND OUTLOOK

In one sense neural networks are nonlinear models having a methodology of their own. From a spatial analysis point of view neural networks can generally be used anywhere one would ordinarily use a linear or nonlinear specification, with estimation proceeding via appropriate techniques. The now rather well-developed theory of estimation of misspecified models applies immediately to provide appropriate interpretations and inferential procedures.

Neural networks have essentially a broader utility that has yet to be fully appreciated by spatial analysts, but which has the potential to significantly enhance scientific understanding of spatial phenomena and spatial processes subject to neural network modelling. In particular, the estimates obtained from neural network learning may serve as a basis for formal statistical inference, making possible statistical tests of specific hypotheses of interest. Because of the ability of neural networks to extract complex nonlinear effects, the alternatives against which such tests can have power may extend usefully beyond those with the reach of more conventional methods, such as linear models for regression and classification.

Although we have covered a fair amount of ground in this chapter we have only scratched the surface of the modelling possibilities offered by neural networks. To mention some additional models treated in the field of neural networks, we note that competitive learning networks have been much studied with applications, for example, to the travelling salesman problem and remote sensing classification problems, and that radial basis function networks in which the activation for a hidden unit is determined by the distance between the input vector and a prototype vector are also standard objects of investigation. Leung (1997), for example, illustrates the use of radial basis function networks for rule learning. We, moreover, did not consider neural networks for unsupervised feature discovery which in statistical terms correspond to cluster analysis and/or latent structure analysis.

For neural networks to find a place in spatial data analysis they need to overcome their current limitations, mainly due to the relative absence of established procedures for model identification, comparable to those for spatial econometric modelling techniques. In particular, providing tests specifically designed to test the adequacy of neural models is a research issue on its own right. Despite significant improvements in our understanding of the fundamentals of neural network modelling, there are many open problems and directions for future research. From a spatial analytic perspective an important avenue for further investigation is the incorporation of spatial dependency in the network representation that received less attention in the past than it deserves. Another is the application of Bayesian inference techniques to neural networks. A Bayesian approach would provide an alternative framework for dealing with the issues of network complexity and would avoid many of the problems discussed in this chapter. In particular, error bars and confidence intervals can easily be assigned to the predictors generated by neural networks, without the need of bootstrapping.

NOTES

1 Neural networks can model cortical local learning and signal processing, but they are not the

brain, neither are many special purpose systems to which they contribute (Weng and Hwang, 2006).

2 Feedforward neural networks are sometimes also called multilayer perceptrons even though the term perceptron is usually used to refer to a network with linear threshold gates rather than with continuous nonlinearities. Radial basis function networks, recurrent networks rooted in statistical physics, self-organizing systems and ART (Adaptive Resonance Theory) models are other important classes. For a fuzzy ARTMAP multispectral classification see Gopal and Fischer (1997).

3 A generalization of this network architecture is to allow skip-layer connections from input to output, each of which is associated with a corresponding adaptive parameter. But note that a network with sigmoidal hidden units can always mimic skip-layer connections for bounded input values by using sufficiently small single hidden layer weights. Skip-layer connections, however, can be easier to implement and interpret in practice.

4 Networks with closed directed cycles are called *recurrent* networks. There are three types of such networks: *first*, networks in which the input layer is fed back into the input layer itself; *second*, networks in which the hidden layer is fed back into the input layer, and *third*, networks in which the output layer is fed back into the input layer. These feedback networks are useful when input variables represent time series.

5 Note, we could alternatively use product rather than summation hidden units to supplement the inputs to a neural network with higher-order combinations of the inputs to increase the capacity of the network in an information capacity sense. These networks are called *product unit* rather than summation unit networks (see Fischer and Reismann, 2002b).

6 This term should not be confused with the term bias in a statistical sense.

7 The inverse of this function is called link function in the statistical literature. Note that radial basis function networks may be viewed as single hidden layer networks that use radial basis function nodes in the hidden layer. This class of neural networks asks for a two stage approach for training. In the first stage the parameters of the basis functions are determined, while in the second stage the basis functions are kept fixed and the second layer weights are found (see Bishop, 1995: 170 pp.).

8 This is the same idea as incorporating the constant term in the design matrix of a regression by inserting a column of ones.

9 In some cases there may be some practical advantage to use a *tanh* function instead. But note that this leads to results equivalent to the logistic function.

10 This viewpoint directs attention to the literature on numerical optimization theory, with particular reference to optimization techniques that use higher-order information such as conjugate gradient procedures and Newton's method. The methods use the gradient vector (first-order partial derivatives) and/or the Hessian matrix (second-order partial derivatives) of the error function to perform optimization, but in different ways. A survey of first- and second-order optimization techniques applied to network training can be found in Cichocki and Unbehauen (1993).

11 When using an iterative optimization algorithm, some choice has to be made of when to stop the training process. There are various criteria that may be used. For example, training may be stopped when the error function or the relative change in the error function falls below some prespecified value.

12 The particular form of $\eta(\tau)$ most commonly used is described by $\eta(\tau) = c/\tau$ where c is a small constant. Such a choice is sufficient to guarantee convergence of the stochastic approximation algorithm (Ljung, 1977).

13 The term *backpropagation* is used in the literature to mean very different things. Sometimes, the feedforward neural network architecture is called a backpropagation network. The term is also used to describe the training of a feedforward neural network using gradient descent optimization applied to a sum-of-squares error function.

14 Note that limited data sets make the determination of H more difficult if there is not enough data available to hold out a sufficiently large independent test sample.

15 A neural network is said to be overfitted to the data if it obtains an excellent fit to the training data, but gives a poor representation of the unknown function which the neural network is approximating.

16 In conventional curve fitting, the use of this regularizer is termed *ridge regression*.

17 Note that a reliable pseudo-random number generator is essential for the valid application of the bootstrap approach.

REFERENCES

AQ:Please check as many references are not supplied in the text.

Anders, U. and Korn, O. (1999). Model selection in neural networks. *Neural Networks*, **12**(2): 309–323.

Baldi, P. and Chauvin, Y. (1991). Temporal evolution of generalization during learning in linear networks. *Neural Computation*, **3**(4): 589–603.

Bäck, T., Fogel, D.B. and Michaelewicz, Z. (eds) (1997). *Handbook of Evolutionary Computation*. New York and Oxford: Oxford University Press.

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

Bishop, C.M. (2006). *Pattern Recognition and Machine Learning*. New York: Springer.

Breiman, L. (1996). Heuristics of instability and stabilization in model selection. *The Annals of Statistics*, **24**(6): 2350–2383.

Bridle, J.S. (1994). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In: Fogelman Soulié, F. and Hérault, J. (eds), *Neurocomputing. Algorithms, Architectures and Applications*. pp. 227–236. Berlin, Heidelberg and New York: Springer.

Carpenter, G.A. (1989). Neural network models for pattern recognition and associative memory. *Neural Networks*, **2**(4): 243–257.

Carpenter, G.A., Grossberg, S. and Reynolds, J.H. (1991). ARTMAP supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural Networks*, **4**(5): 565–588.

Cichocki, A. and Unbehauen, R. (1993). *Neural Networks for Optimization and Signal Processing*. Chichester: Wiley.

Corne, S., Murray, T., Openshaw, S., See, L. and Turton, I. (1999). Using computational intelligence techniques to model subglacial water systems. *Journal of Geographical Systems*, **1**(1): 37–60.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control Signals and Systems*, **2**: 303–314.

Efron, B. (1982). *The Jackknife, the Bootstrap and Other Resampling Plans*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Efron, B. and Tibshirani, R. (1993). *An Introduction to the Bootstrap*. New York: Chapman and Hall.

Finnoff, W. (1991). Complexity measures for classes of neural networks with variable weight bounds. *Proceedings of the International Geoscience and Remote Sensing Symposium* (IGARSS'94, Volume 4), pp. 1880–1882. Piscataway, NJ: IEEE Press.

Finnoff, W., Hergert, F. and Zimmerman, H.G. (1993). Improving model selection by nonconvergent methods. *Neural Networks*, **6**(6): 771–783.

Fischer, M.M. (1998). Computational neural networks – A new paradigm for spatial analysis. *Environment and Planning*, **A30**(10): 1873–1892.

Fischer, M.M. (2000). Methodological challenges in neural spatial interaction modelling: the issue of model selection. In: Reggiani, A. (ed.), *Spatial Economic Science*: *New Frontiers in Theory and Methodology*. pp. 89–101. Berlin, Heidelberg and New York: Springer.

Fischer, M.M. (2002). Learning in neural spatial interaction models: A statistical perspective', *Journal of Geographical Systems,* **4**(3): 287–299.

Fischer, M.M. (2005). Spatial analysis. In Longley, P., Goodchild, M.F., Maguire, D.J. and Rhind, D.W. (eds), *Geographical Information Systems. Principles, Techniques, Management and Applications. Second Edition, Abridged*, (CD-ROM). Hoboken, New Jersey: Wiley.

Fischer, M.M. (2006a). Neural networks. A general framework for non-linear function approximation. *Transactions in GIS*, **10**(4): 521–533.

Fischer, M.M. (2006b). *Spatial Analysis and Geocomputation. Selected Essays*. Berlin, Heidelberg and New York: Springer.

Fischer, M.M. and Getis, A. (eds) (1997). *Recent Developments in Spatial Analysis. Spatial Statistics, Behavioural Modelling, and Computational Intelligence*. Berlin, Heidelberg and New York: Springer.

Fischer, M.M. and Gopal, S. (1994). Artificial neural networks: A new approach to modelling interregional telecommunication flows. *Journal of Regional Science,* **34**(4): 503–527.

Fischer, M.M. and Leung, Y. (1998). A genetic-algorithm based evolutionary computational neural network for modelling spatial interaction data. *The Annals of Regional Science,* **32**(3): 437–458.

Fischer, M.M. and Leung, Y. (eds) (2001). *GeoComputational Modelling: Techniques and Applications*. Berlin, Heidelberg and New York: Springer.

Fischer, M.M. and Reismann, M. (2002a). Evaluating neural spatial interaction modelling by bootstrapping. *Networks and Spatial Economics,* **2**(3): 255–268.

Fischer, M.M. and Reismann, M. (2002b). A methodology for neural spatial interaction modeling. *Geographical Analysis,* **34**(2): 207–228.

Fischer, M.M. and Staufer, P. (1999). Optimization in an error backpropagation neural network environment with a performance test on a spectral pattern classification problem. *Geographical Analysis*, **31**(2): 89–108.

Fischer, M.M., Hlaváckova-Schindler, K. and Reismann, M. (1999). A global search procedure for parameter estimation in neural spatial interaction modelling. *Papers in Regional Science,* **78**(2): 119–134.

Fischer, M.M., Reismann, M. and Hlaváckova-Schindler, K. (2003). Neural network modelling of constrained spatial interaction flows: Design, estimation and performance issues. *Journal of Regional Science,* **43**(1): 35–61.

Fischer, M.M., Gopal, S., Staufer, P. and Steinnocher, K. (1997). Evaluation of neural pattern classifiers for a remote sensing application. *Geographical Systems,* **4**(2): 195–226.

Fogel, D.B. (1995). *Evolutionary Computation*: *Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ: IEEE Press.

Fogel, D.B. and Robinson, C.J. (eds) (1996). *Computational Intelligence*. Piscataway: IEEE Press and Wiley-Interscience.

Foody, G.M., and Boyd, D.S. (1999). Fuzzy mapping of tropical land cover along an environmental gradient from remotely sensed data with an artificial neural network. *Journal of Geographical Systems*, **1**(1): 23–35.

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

Gahegan, M. (2000). On the application of inductive machine learning tools to geographical analysis. *Geographical Analysis*, **32**(1): 113–133.

Gahegan, M., German, G. and West, G. (1999). Improving neural network performance on the classification of complex geographic datasets. *Journal of Geographical Systems*, **1**(1): 3–22.

Goldberg, D.E. (1989). *Genetic Algorithms*. Reading, MA: Addison-Wesley.

Gopal, S. and Fischer, M.M. (1996). Learning in single hidden-layer feedforward network models. *Geographical Analysis* **28**(1): 38–55.

Gopal, S. and Fischer, M.M. (1997). Fuzzy ARTMAP – a neural classifier for multispectral image classification. In: Fischer, M.M. and Getis, A. (eds), *Recent Developments in Spatial Analysis*, pp. 306–335. Berlin, Heidelberg and New York: Springer.

Grossberg, S. (1988). Nonlinear neural networks. Principles, mechanisms and architectures. *Neural Networks*, **1**(1): 17–61.

Hassoun, M.H. (1995). *Fundamentals of Artificial Neural Networks*. Cambridge, MA and London, England: MIT Press.

Hastie, T., Tibshirani, R. and Friedman, J. (2001). *The Elements of Statistical Learning*. Berlin, Heidelberg and New York: Springer.

Haykin, S. (1994). *Neural Networks. A Comprehensive Foundation*. New York: Macmillan College Publishing Company.

Hertz, J., Krogh, A. and Palmers, R.G. (1991). *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison-Wesley.

Hlaváckova-Schindler, K. and Fischer, M.M. (2000). An incremental algorithm for parallel training of the size and the weights in a feedforward neural network. *Neural Processing Letters*, **11**(2): 131–138.

Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, **2**(5): 359–368.

Huang, G.-B. and Siew, C.-K. (2006). Real-time learning capability of neural networks. *IEEE Transactions on Neural Networks*, **17**(4): 863–878.

Janson, D.J. and Frenzel, J.F. (1993). Training product unit neural networks with genetic algorithms. *IEEE Expert*, **8**(5): 26–33.

Kohonen, T. (1988). *Self-Organization and Associative Memory*. Berlin, Heidelberg and New York: Springer.

Kuan, C.-M. and White, H. (1991). Artificial neural networks: An econometric perspective. *Econometric Reviews*, **13**(1): 1–91.

Kůrková, V. (1992). Kolmogorov's theorem and multilayer neural networks. *Neural Networks*, **5**(3): 501–506.

Leung, K.-S., Ji, H.-B. and Leung, Y. (1997). Adaptive weighted outer-product learning associative memory. *IEEE Transactions on Systems, Man, and Cybernetics* – Part B, **27**(3): 533–543.

Leung, Y. (1997). Feedforward neural network models for spatial data classification and rule learning. In: Fischer, M.M. and Getis, A. (eds), *Recent Developments in Spatial Analysis*. pp. 289–305. Berlin, Heidelberg and New York: Springer.

Leung, Y. (2001). Neural and evolutionary computation methods for spatial classification and knowledge acquisition. In: Fischer, M.M. and Leung, Y. (eds), *GeoComputational Modelling. Techniques and Applications*. pp. 71–108. Berlin, Heidelberg and New York: Springer.

Leung, Y., Chen, K.-Z. and Gao, X.-B. (2003). A high-performance feedback neural network for solving convex nonlinear programming problems. *IEEE Transactions on Neural Networks* **14**(6): 1469–1477.

Leung, Y., Dong, T.-X. and Xu, Z.-B. (1998). The optimal encodings for biased association in linear associative memory. *Neural Networks* **11**(5): 877–884.

Leung, Y., Gao, X.-B. and Chen, K.-Z. (2004). A dual neural network for solving entropy-maximising models. *Environment and Planning*, **A36**(5): 897–919.

Leung, Y., Chen, K.-Z., Jiao, Y.-C., Gao, X.-B. and Leung, K.S. (2001). A new gradient-based neural network for solving linear and quadratic programming problems. *IEEE Transactions on Neural Networks*, **12**(5): 1074–1083.

Ljung, L. (1977). Analysis of recursive stochastic algorithms. *IEEE Transactions on Automatic Control*, **AC-22**: 551–575.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, **5**: 115–133.

Mineter, M.J. and Dowers, S. (1999). Parallel processing for geographical applications: A layered approach. *Journal of Geographical Systems*, **1**(1): 61–74.

Moody, J.E. (1992). The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In: Moody, J.E., Hanson, S.J. and Lippman, R.P. (eds), *Advances in Neural Information Processing Systems 4*, pp. 683–690. San Mateo, CA: Morgan Kaufmann.

Murata, N. Yoshizawa, S. and Amari, S. (1993). Learning curves, model selection and complexity of neural networks. In: Hanson, S.J., Cowan, J.D. and Giles, C.L. (eds), *Advances in Neural Information Processing Systems 5*, pp. 607–614. San Mateo, CA: Morgan Kaufmann.

Nocedal, J. and Wright S.J. (1999). *Numerical Optimization*. Berlin, Heidelberg and New York: Springer.

Openshaw, S. (1993). Modelling spatial interaction using a neural net. In: Fischer, M.M. and Nijkamp, P. (eds), *GIS, Spatial Modelling, and Policy*. pp. 147–164. Berlin, Heidelberg and New York: Springer.

Openshaw, S. (1994). Neuroclassification of spatial data. In: Hewitson, B.C. and Crane, R.G. (eds), *Neural Nets: Applications in Geography*. pp. 53–70. Boston: Kluwer Academic Publishers.

Openshaw, S. and Abrahart, R.J. (eds) (2000). *GeoComputation*. London and New York: Taylor & Francis.

Openshaw, S. and Openshaw, C. (1997). *Artificial Intelligence in Geography*. Chichester: Wiley.

Plutowski, M., Sakata, S. and White, H. (1994). Cross-validation estimates IMSE. In: Cowan, J.D., Tesauro, G. and Alspector, J. (eds), *Advances in Neural Information Processing Systems 6*, pp. 391–398. San Francisco: Morgan Kaufmann.

Poggio, T. and Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, **78**(9): 91–106.

Press, W.H., Teukolky, S.A., Vetterling, W.T. and Flannery, B.P. (1992). *Numerical Recipes in C. The Art of Scientific Computing*. 2nd edn. Cambridge: Cambridge University Press.

Ripley, B.D. (1994). Neural networks and related methods for classification (with discussion). *Journal of the Royal Statistical Society B*, **56**(3): 409–456.

Ripley, B.D. (1996). *Pattern Recognition and Neural Networks*. Cambridge: Cambridge University Press.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Washington D.C.: Spartan Books.

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

Schwefel, H.-P. (1994). *Evolution and Optimum Seeking*. New York: Wiley.

Specht, D.F. (1991). A general regression neural network. *IEEE Transactions on Neural Networks*, **2**(6): 568–576.

Stepniewski, W. and Keane, J. (1997). Pruning backpropagation neural networks using modern stochastic optimization techniques. *Neural Computing and Applications*, **5**(2): 76–98.

Tibshirani, R. (1996a). A comparison of some error estimates for neural network models. *Neural Computation*, **8**(1): 152–163.

Tibshirani, R. (1996b). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society*, **B58**: 267–288.

Wedge, D., Ingram, D., McLean, D., Mingham, C. and Bandar, Z. (2006). On global-local artificial neural networks for function approximation. *IEEE Transactions on Neural Networks*, **17**(4): 942–952.

Weigend, A.S., Rumelhart, D.E. and Huberman, B.A. (1991). Generalization by weight elimination with application to forecasting. In: Lippman, R., Moody, J. and Touretzky, D. (eds), *Advances in Neural Information Processing Systems* 3, pp. 875–882. San Mateo, CA: Morgan Kaufmann.

Weng, J. and Hwang, W.-S. (2006). From neural networks to the brain: Autonomous mental development. *IEEE Computational Intelligence Magazine*, **1**(3): 15–31.

White, H. (1989). Learning in artificial neural networks: a statistical perspective. *Neural Computation*, **1**(4): 425–464.

White, H. (1992). *Artificial Neural Networks. Approximation and Learning Theory*. Oxford, UK and Cambridge, USA: Blackwell.

White, H. and Racine, J. (2001). Statistical inference, the bootstrap, and neural-network modeling with applications to foreign exchange rates. *IEEE Transactions on Neural Networks*, **12**(4): 657–673.

Widrow, B. and Hoff, M.E. Jr. (1960). Adaptive switching circuits. *IRE Western Electric Show and Convention Record*, **Part 4**: 96–104.

Wilkinson, G.G. (1997). Neurocomputing for earth observation – recent developments and future challenges. In: Fischer, M.M. and Getis, A. (eds), *Recent Developments in Spatial Analysis*, pp. 289–305. Berlin, Heidelberg and New York: Springer.

Wilkinson, G.G. (2001). Spatial pattern recognition in remote sensing by neural networks. In: Fischer, M.M. and Leung, Y. (eds), *GeoComputational Modelling. Techniques and Applications*. pp. 145–164. Berlin, Heidelberg and New York: Springer.

Wilkinson, G.G., Fierens, F. and Kanellopoulos, I. (1995). Integration of neural and statistical approaches in spatial data classification. *Geographical Systems*, **2**(1): 1–20.

Yao, X. (1996). A review of evolutionary artifical neural networks. *International Journal of Intelligent Systems*, **8**(4): 539–567.

Yao, X. (2001). Evolving computational neural networks through evolutionary computation. In: Fischer, M.M. and Leung, Y. (eds), *GeoComputational Modelling. Techniques and Applications*. pp. 35–70. Berlin, Heidelberg and New York: Springer.

Yao, X., Fischer, M.M. and Brown, G. (2001). Neural network ensembles and their application to traffic flow prediction in telecommunication networks. In: *Proceedings of the 2001 IEEE-INNS-ENNS International Joint Conference on Neural Networks*. pp. 693–698. Piscataway, NJ: IEEE Press.

Zapranis, A. and Refenes, A.-P. (1998). *Principles of Neural Model Identification, Selection and Adequacy. With Applications to Financial Econometrics*. London: Springer.

Zhang, G., Patuwo, B.E. and Hu, M.Y. (1998). Forecasting with artificial neural networks: The state of the art. *International Journal of Forecasting*, **14**(1): 35–62.