Neural networks. A class of flexible non-linear models for regression and classification

Fischer, Manfred M.

Published in:
Handbook of Research Methods and Applications in Economic Geography

Published: 01/01/2015

Document Version
Peer reviewed version

Neural networks. A class of flexible non-linear models for regression and classification*

Manfred M. Fischer

1 Introduction

Neural networks form a field of research that has enjoyed rapid expansion and increasing popularity in recent years. The exuberance of this growth has been accompanied by exaggerated claims concerning the technological potential of neural networks. In addition, a definite mystique perceived by those outside the field arises from the origins of neural networks in the study of natural neural systems, and in the associated metaphorical jargon in the field. Both the exaggerated claims and the mystique may have acted to lessen the amount of serious attention given to neural networks in economic geography and regional science.

This chapter is intended as a convenient resource for those interested in a more fundamental view of the neural network modelling approach. The primary aim is to discuss some issues that are crucial for the design and understanding of neural network models, with a strong emphasis on their practical use for solving regression and classification problems. A regression problem occurs when the goal of analysis is to predict a continuous variable given the values of an M-dimensional vector, say x, of

* to be published in Karlsson C. and Andersson M. (eds) (2012): *Handbook of Research Methods and Applications in Economic Geography*. Edward Elgar

input variables. A classification problem arises when an object of interest needs to be assigned into one of several predefined classes based on a number of observed attributes related to that object. Note that many problems in economic geography and regional science can be treated as classification problems. Examples include the classification of tourists into market segments, the assignment of dwellings to housing submarkets, and the classification of parcells to land use categories, among many others.

We can not do justice to the entire spectrum of neural network models. Instead, attention is focused on a particular class of neural networks that have shown to be of greatest practical value, namely the class of feeedforward neural networks. We use statistical arguments to gain important insights into the problems and properties of this modelling approach that are important for successful applications. Due to space limitations, no attempt has been made to illustrate the discussion with empirical examples.

The attractiveness of feedforward neural networks is due to two characteristics. First, they are devices for non-parametric statistical inference. No particular structure or parametric form is assumed *a priori*. This is particularly useful in the cases of regression and classification problems where solutions require knowledge that is difficult to specify a priori, but for which there are sufficient observations. Second, feedforward neural networks provide a very flexible framework to approximate arbitrary non-linear mappings from a set of input variables to a set of output variables, where the form of mapping is governed by a number of adjustable parameters, called weights. In the case of regression problems, it is the regression function that we wish to approximate. The network outputs are the explanatory variables, the weights the regression parameters, and the network output is the dependent variable. From the

statistician's point of view, feedforward neural networks are non-parametric, non-linear regression models, which can also be used to classify via regressions.

The chapter is organized as follows. We begin by considering the functional form of feedforward neural network models, including the specific parameterization of the activation functions that form the basis for neural network models. We then discuss the problem of determining the network parameters within a maximum likelihood framework that involves the solution of a non-linear optimization problem. This requires the evaluation of derivatives of the log-likelihood function with respect to the network parameters, and we discuss how these can be obtained efficiently for regression and classification problems, using the technique of error backpropagation.

We continue to consider the issue of network complexity and briefly discuss regularization and early stopping in network training as approaches to optimizing the complexity of a network model (complexity measured in terms of the number of hidden units) in order to achieve the best predictive performance. The next section moves attention to the issue of how to appropriately test the predictive performance of a neural network model. Some conclusions 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 textbooks with introductory character, of which Hertz *et al*. (1991), Ripley (1996) and Bishop (2006) appear to be most suitable for the audience of this Handbook. Readers interested in spatial interaction or flow data analysis are referred to Fischer and Reismann (2002b) to find a useful methodology for neural spatial interaction modelling.

2 Feedforward network functions

Feedforward neural networks consist of nodes (also known as processing units or simply units) that are organized in layers. Figure 1 shows a schematic diagram of a typical feedforward neural network that contains an intermediate layer of processing units. Intermediate layers of this sort are called *hidden* layers to distinguish them from the input and output layers. In this network there are *M* input nodes representing input variables $x_1,..., x_M$; *H* hidden units representing hidden variables $z_1,..., z_H$; and *K* output units representing output variables $y_1,..., y_K$. Weight parameters are represented by links between the nodes. The bias parameters (not to be confused with the statistical conecpt of bias) are denoted by links coming from additional input and hidden variables, say, x_0 and z_0 where values are permanently set at one so that $x_0 = 1$ and $z_0 = 1$. Observe the feedforward structure in Fig. 1 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.

Figure 1 about here

Figure 1. Network diagram for the two-layer neural network model corresponding to Eq. (6); normally all units in one layer are connected to all in the next layer, as shown. The input, hidden and output variables are represented by nodes, and the weight parameters by links between the nodes. The bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. The arrows represent the direction of information flow through the network during the forward propagation.

The network architecture shown in Fig. 1 is the most commonly used in practice. The term network architecture refers to the topological arrangement of nodes. Note that there is some confusion in the literature concerning the terminology for counting the number of layers in such networks. The network in Fig. 1 may be described as a 3-layer network which counts the number of layers of units, and treats the inputs as units, or as single-hidden-layer network which counts the number of layers of hidden units. We recommend a terminology in which Fig. 1 is called a two-layer network since it is the number of layers of adaptive weights that is important for determining the network properties.

Any network diagram can be converted into a corresponding mapping function, provided that the diagram is feedforward as in Fig. 1 so that it does not contain closed directed cycles. This guarantees that the network model can be described by a series of functional transformations as follows. First, we construct linear combinations of the input variables $x_1, ..., x_M$ in the form

$$a_h = \sum_{m=1}^{M} w_{hm}^{(1)} x_m + w_{ho}^{(1)} \tag{1}$$

where $h = 1, ..., H,$ and the superscript (1) indicates that the corresponding parameters are in the first layer of the network. We shall refer to the parameters $w_{hm}^{(1)}$ as (*connection*) *weights* and the parameters $w_{ho}^{(1)}$ as *biases*. The quantities $a_h (h = 1, ..., H)$ are known as hidden *activations*. Each of them is then transformed using a continuous and differentiable, non-linear activation function $\phi(\bullet)$ to give

$$z_h = \phi(a_h). \tag{2}$$

These quantities are called *hidden units*. They are again linearly combined to generate output activations

$$a_k = \sum_{h=1}^{H} w_{kh}^{(2)} z_h + w_{ko}^{(2)} \tag{3}$$

where $k = 1, ..., K,$ and K is the total number of outputs. This transformation corresponds to the second layer of the network, and the $w_{ko}^{(2)}$ are bias parameters.

Finally, the output unit activations are transformed using an appropriate activation function $\psi(\bullet)$ to produce a set of network outputs $y_k (k = 1, ..., K)$

$$y_k = \psi(a_k). \tag{4}$$

Combining these stages of transformation yields the overall network function that takes the form

$$y_k(x,w) = \psi\left(\sum_{h=1}^{H} w_{kh}^{(2)}\ \phi\left(\sum_{m=1}^{M} w_{hm}^{(1)}\ x_m + w_{ho}^{(1)}\right) + w_{ko}^{(2)}\right) \tag{5}$$

for $k = 1, ..., K$. w represents a vector of all the weights and bias parameters, and $x = (x_1, ..., x_M)'$. It is convenient to absorb the bias parameters, $w_{h0}^{(1)}$ $(h = 1, ..., H)$, into the set of first-layer weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$ so that Eq. (1) takes the form $\alpha_h = \sum_{m=0}^{M} w_{hm}^{(1)} x_m$. Similarly the second layer biases, $w_{k0}^{(2)}$ $(k = 1, ..., K)$, can be absorbed into the set of second layer weights, so that the overall network function becomes

$$y_k(x,w) = \psi\left(\sum_{h=0}^{H} w_{kh}^{(2)}\ \phi\left(\sum_{m=0}^{M} w_{hm}^{(1)}\ x_m\right)\right). \tag{6}$$

The numbers of input and output units, *M* and *K*, are generally determined by the dimensionality of the data set and/or the problem under study, while the number *H* of hidden units is a free parameter that controls the number of weights and biases in the network. Note that Eq. (6) can easily be generalized to allow different output and hidden units to have individual activation functions ψ_k $(k = 1, ..., K)$ and ϕ_h $(h = 1, ..., H)$. For simplicity we assume, however, that $\psi_k = \psi$ for all *k* and $\phi_h = \phi$ for all *h*.

Neural network models of type (6) are rather general. They provide a flexible way to parameterize a fairly general non-linear function from a set of input variables

$\{x_m: m = 1, ..., M\}$ to a set of output variables $\{y_k: k = 1, ..., K\}$ controlled by a *Q*-dimensional vector *w* of adjustable parameters, where $Q = H(M + 1) + K(H + 1)$.

The approximation capabilities of feedforward neural network models have been widely studied (see, for example, Cybenko 1989, Hornik et al. 1989, Stinchcombe and White 1989) and found to be very general. Such feedforward neural network models, with more or less general types of activation functions ϕ and ψ, are said to be *universal approximators*. They can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy, by increasing the size of the hidden layer (that is *H*). Because of this approximation property one hidden layer is sufficient for regression and classification problems, and thus we restrict our attention to single-hidden-layer networks. Note, however, that it may be more parsimonous to use fewer hidden units in two or more hidden layers.

This result holds for a wide range of non-linear hidden layer activation functions as long as they are continuous and differentiable. But they are generally chosen to be sigmoidal functions (the term sigmoidal means S-shaped) such as the logistic sigmoid function

$$\phi(a_h) = [1 + \exp(-a_h)]^{-1} \tag{7}$$

where outputs lie in the range (0, 1). The function satisfies the following symmetry property: $\phi(-a_h) = 1 - \phi(a_h)$. Equivalently, one can use the tanh function because this is related to the logistic sigmoid by $\tanh(a_h) = 2\phi(a_h) - 1$.

The choice of the output unit activation function $\psi(\bullet)$ is determined by the nature of the data and the assumed distribution of target variables (Bishop 2006, pp. 227-228). For regression problems, it has been conventional to take $\psi(\bullet)$ to be a sigmoidal function of its net input. But in situations in which the assumption of normally distributed noise in the observations holds the identity function is more appropriate so that $y_k = a_k$.

For binary classification problems – that is, for problems in which the goal is to provide a binary classification of each input vector for each of several classes (called multiple classification problems), each output unit activation function is transformed using a logistic sigmoid function so that

$$y_k = [1 + \exp(-a_k)]^{-1} \tag{8}$$

while for standard multiclass classification problems (that is, $K > 2$ classes) in which each input is assigned to one of K mutually exclusive classes (categories) gives rise to the softmax activation function (see Bridle 1989)

$$y_k = \psi(a_k) = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)} \tag{9}$$

where $0 \le y_k \le 1$ and $\sum_{k=1}^{K} y_k = 1$. The activation function is known as the normalized exponential function and can be regarded as a multiclass generalization of the logistic sigmoid.

A neural network with a single logistic output unit can be seen as a non-linear extension of logistic regression. With many logistic units, it corresponds to linked logistic regressions of each class versus the others. If the activation functions of the output units in a network are taken to be linear, we have a standard linear model augmented by non-linear terms. Given the popularity of linear models in data analysis, this form is particularly appealing, as it suggests that neural network models can be viewed as extensions of – rather than as alternatives to – familiar statistical models such as linear regression models, discriminant analysis and cluster analysis. The hidden unit activations can then be considered as latent variables whose inclusion enriches the linear model.

3 Network training and the error function

Up to now, we have considered neural networks as a general class of parametric non-linear functions from a *M*-dimensional vector of input variables $x_1,...,x_M$ to a *K*-dimensional vector of output variables $y_1,...,y_K$, controlled by a *Q*-dimensional vector of adjustable network parameters $w_1, ..., w_Q$. The process of determining these weights and bias parameters is called *network training* (also known as network learning) and involves adjusting the parameters while fixing the topology (that is *K*, *H* and *M*) and the activation functions ϕ and ψ.

Suppose that the set U_N of *N* input-output pairs [called training set]

$$U_N = \{(x_n, t_n): n = 1, ..., N\} = \{(x_{n1}, x_{n2}, ..., x_{nM}; t_{n1}, t_{n2}, ..., t_{nK}): n = 1, ..., N\} \quad (10)$$

comprises N independent observations on the M input variables x_m $(m = 1, ..., M)$ and the associated K-dimensional target output vector, denoted by $t = (t_1, ..., t_K)'$. Then network training strives to finding a parameter vector w which minimizes an error function $E(w)$ that measures the misfit between the neural network model $y(x, w)$, for any given w, and the training set data points.

The error functions we consider can be motivated from the principle of maximum likelihood (see Bishop 1995, pp. 195-197). For the set of training data $\{(x_n, t_n): n = 1, ..., N\}$ the likelihood can be written as

$$L(w) = \prod_{n=1}^{N} p(x_n, t_n) = \prod_{n=1}^{N} p[t_n \mid y(x_n, w)]\ p(x_n) \quad (11)$$

since we have assumed that each data point (x_n, t_n) is drawn independently from the same distribution (perhaps not altogether a realistic assumption in the case of spatial data), and thus we can multiply the probabilities. It is generally more convenient to minimize the negative logarithm of the likelihood rather than maximizing the likelihood since the negative logarithm is a monotonic function, and these are equivalent procedures.

$$E(w) = -\ln L(w) = -\sum_{n=1}^{N} \ln p[t_n \mid y(x_n, w)] - \sum_{n=1}^{N} \ln p(x_n) \quad (12)$$

where *E* is called an *error function*, also known as loss function. The second term on the right hand side of Eq. (12) does not depend on the network parameters, and hence represents an additive constant that can be dropped from the error function so that

$$E(w) = -\sum_{n=1}^{N} \ln p\left[t_n \mid y(x_n, w)\right]. \tag{13}$$

Note that the error function takes the form of a sum over patterns of an error term for each pattern (x_n, t_n) separately. This follows from the assumed independence of the data points under the given distribution. Different choices of error function arise from different assumptions about the form of the conditional distribution $p(t, x)$. For regression problems, the targets *t* consist of continuous (real-valued) quantities whose values we are attempting to predict, while for classification problems they represent labels defining class membership or, more generally, estimates of the probabilities of class membership.

The error function for regression problems

We start by discussing regression problems, and assume that the *K* target variables are independent conditional on *x* and *w* with shared noise precision α, then the conditional distribution is given by a Gaussian

$$p(t \mid x, w) = \mathcal{N}(t \mid y(x, w), \alpha^{-1} I). \tag{14}$$

where α is the precision (inverse variance) of the Gaussian noise and *I* is the *K*-by-*K* identity matrix. For the conditional distribution given by Eq. (14) it is sufficient to take the output unit activation $\psi(\bullet)$ to be the identity, because such a neural network model can approximate any continuous function from *x* to *y*.

Given a data set of *N* independent, identically distributed observations $\{(x_n, t_n): n = 1, ..., N\}$ on *M* input and *K* output variables, we obtain the error function (see Bishop 2006, p. 233)

$$E(w) = \frac{\alpha}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2 - \frac{NK}{2} \ln \alpha + \frac{NK}{2} \ln(2\pi) \tag{15}$$

which can be used to determine the parameters *w* and α. Let us consider first the determination of *w* and note that for the purpose of error minimization, the second and third terms on the right hand side of Eq. (15) are independent from *w* and hence can be discarded. Similarly, the overall factor α in the first term can be omitted. Thus, we obtain the familiar expression for the sum-of-squares error function given by

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \{y_k(x_n, w) - t_{nk}\}^2 = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2 . \tag{16}$$

The value of *w* that minimizes $E(w)$ will be denoted by w_{ML} because it corresponds to the maximum likelihood solution. In practice, however, the non-linearity of the network

model function $y(x_n, w)$ causes the error $E(w)$ to be non-convex so that local minima of the error function corresponding to the local maxima of the likelihood may be found.

Having found w_{ML}, the optimal value for α can be found by minimization of E given by Eq. (15) with respect to α. The noise precision is then given by

$$\frac{1}{\alpha_{ML}} = \frac{1}{NK} \sum_{n=1}^{N} \sum_{k=1}^{K} \{ y_k(x_n, w_{ML}) - t_{nk} \}^2 \tag{17}$$

where K is the number of target variables. Equation (17) says that the optimal value of α is proportional to the residual value of the sum-of-squares error function at its minimum (see Bishop 2006, p. 234).

Note that there is a natural pairing of the error function, given by the negative log-likelihood and the output unit activation function $\psi(\bullet)$. In the regression case (under the Gaussian assumption of error), we can view the network model as having an output activation function ψ that is the identity, so that $y_k = a_k$. The corresponding sum-of-squares error function then has the characteristic that

$$\frac{\partial E}{\partial a_k} = y_k - t_k . \tag{18}$$

This property will be used when discussing the technique of error backpropagation in one of the subsequent sections.

The error functions for classification problems

For regression problems, the target variable *t* is simply the vector of real numbers whose values we wish to predict. In the case of classification, there are various ways to use target values to represent class labels. The most convenient way in the case of two-class problems is the binary representation in which there is a single target variable $t \in \{0,1\}$ such that $t = 1$ represents class C_1 and $t = 0$ class C_2. We can interpret the value of *t* as the probability that the class is C_1, with the probability taking only the extreme values of zero and one.

For $K > 2$ classes, it is convenient to use a 1-of-*K* coding scheme in which *t* is a vector of length *K* such that if the class is C_j, then all elements t_k of *t* are zero except t_j, which takes the value one. For example, if we have $K = 6$ classes, then a pattern from class 3 would be given the target vector $t = (0, 0, 1, 0, 0, 0)'$. Again, we can interpret the value of t_k as the probability that the class is C_k.

Let us consider first, the *case of binary classification*. We consider a network model with a single output whose output activation function is a logistic sigmoid function of the type given by Eq. (7) so that $0 \le y(x, w) \le 1$, and we can interpret $y(x, w)$ as the conditional probability $p(C_1, x)$, with $p(C_2, x)$ given by $1 - y(x, y)$. The conditional probability of targets given inputs is then a Bernoulli distribution of the form

$$p(t \mid x, w) = y(x, w)^t \left\{1 - y(x, w)\right\}^{1-t}. \tag{19}$$

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

$$E(w) = -\sum_{n=1}^{N} \{ t_n \ln y(x_n, w) + (1 - t_n) \ln [1 - y(x_n, w)] \}. \tag{20}$$

Note that there is no analogue of the noise precision α because the target values are assumed to be correctly labeled (Bishop 2006, p. 235).

Let us move next to the case of *K separate binary classifications*. In this case a neural network model with *K* logistic sigmoid output units is an appropriate choice. A binary class label $t_k \in \{0,1\}$, where $k = 1, ..., K,$ is associated with each output. If we assume that the class labels are independent, given the input vector *x*, the conditional distribution of targets *t* is

$$p(t \mid x, w) = \prod_{k=1}^{K} y_k(x, w)^{t_k} [1 - y_k(x, w)]^{1 - t_k}. \tag{21}$$

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

$$E(w) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \{ t_{kn} \ln y_k(x_n, w) + (1 - t_{nk}) \ln [1 - y_k(x_n, w)] \}. \tag{22}$$

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 given by Eq. (18) as in the regression case.

Finally let us consider the *standard multiclass classification problem*, where each input is assigned to one of *K* mutually exclusive classes. In this case, we can use a neural network model with *K* output units each of which has a softmax output activation function defined by Eq. (9) that satisfies $0 \le y_k \le 1$ and $\sum_{k=1}^{K} y_k = 1$. 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 $y_k(x,w) = p(t_k = 1 | x)$. This leads to the error function called the *cross-entropy* error function for the multiclass classification problem

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln\left[y_k(x_n, w)\right]. \tag{23}$$

Once again, the derivative of this error function with respect to the activation for a particular output unit *k* takes the familiar form given by Eq. (8). It is worth noting that in the case of $K = 2$ we can use a network model with a single logistic sigmoid output, alternatively to a network model with two softmax output activations.

In summary, there is a natural pairing of the choice of the output unit activation function and the choice of the error function, according to the type of problem that has to be solved. For regression problems we take linear outputs along with a sum-of-squares error function, for (multiple independent) binary classification problems we use logistic sigmoid outputs with the cross-entropy error function, and for multiclass classification problems softmax outputs with the multiple-class cross-entropy error

function. For classification problems involving only two classes, we can use a single logistic sigmoid output unit, or alternatively we can take a network model with two softmax outputs (Bishop 2006, p. 236).

4 Parameter optimization and local minimization procedures

Learning feedforward neural network weights is like solving an unconstrained, continuous, non-linear optimization problem. The task is to find a weight vector w which minimizes the chosen error function $E(w)$. The problem is, generally, multimodal with multiple local minima.

Since $E(w)$ is a smooth continuous function of w, its smallest value will occur at a point w in the parameter space W such that the gradient ∇ of the error function $E(w)$ vanishes, so that

$$\nabla E(w) = 0. \tag{24}$$

The minimum for which the value of the error function is smallest is called the global minimum, while other minima are called *local minima*. There may be other points that satisfy (24) such as local maxima and saddle points.

Because there is no hope of finding an analytical solution to Eq. (24) one has to resort to numerical iterative procedures. The problem of optimizing a continuous non-linear function is a widely studied problem in the literature, and there exist many

iterative procedures to solve this problem. Examples include gradient descent, Newton and quasi-Newton, and conjugate gradient procedures.

These iterative procedures involve a search through parameter space consisting of a succession of steps of the form

$$w(\tau+1) = w(\tau) + \eta(\tau)\, d(\tau) \tag{25}$$

where $\tau = 0, 1, 2, \ldots$ labels the iteration step. The parameter $\eta(\tau) > 0$ (called the *learning rate* in the neural network literature) determines the length of the step to be taken in the direction of the vector $d(\tau) \simeq \Delta w(\tau)$.

In the numerical optimization literature different techniques are known for the computation of the direction vector $d(\tau)$ (see, for example, Luenberger 1984, Fletcher 1986, Cichocki and Unbehauen 1994). Different procedures involve different choices for the parameter vector increment $\Delta w(\tau)$:

(i) The *gradient descent procedure* is characterized by defining the direction as

$$d(\tau) = -\nabla E[w(\tau)]\,. \tag{26}$$

(ii) For the *Newton's procedure* the search direction is determined by the formula

$$d(\tau) = -\left\{\nabla^2 E[w(\tau)]\right\}^{-1} \nabla E[w(\tau)] \tag{27}$$

where $\nabla^2 E[w(\tau)]$ is the Q-by-Q Hessian matrix of the error function at $w(\tau)$, in which case second derivatives of the error form the elements of the matrix.

(iii) Since the computation of the inverse Hessian matrix may be highly complicated, one may attempt to approximate it by some Q-by-Q symmetric definite matrix $H(\tau)$, that is

$$H(\tau) \simeq \left\{\nabla^2 E[w(\tau)]\right\}^{-1}. \tag{28}$$

In this way we obtain the *quasi-Newton* or *variable metric procedure* in which the search direction is determined as

$$d(\tau) = -H(\tau)\, \nabla E[w(\tau)]. \tag{29}$$

(iv) There are different versions of the *conjugate gradient procedure*. In the Polak-Ribière variant, the search direction is computed as

$$d(\tau) = -\nabla E[w(\tau)] + \beta(\tau)\, d(\tau - 1) \tag{30}$$

for $\tau = 1, 2, ...,$ and

$$d(0) = -\nabla E[w(0)] \tag{31}$$

where $\beta(\tau)$ is a scalar parameter that ensures the sequence of vectors $d(\tau)$ satisfying a mutual conjugacy condition (see Press et al. 1992, pp. 420-422, Fischer and Staufer 1999).

The step length $\eta(\tau)$ in Eq. (25) is usually determined by using *line searches* along the selected search directions. In other words, a positive scalar $\eta = \eta(\tau)$ is sought that minimizes the one-dimensional function

$$E[w(\tau) + \eta\, d(\tau)]. \tag{32}$$

Such minimization procedures determine a local minimum of the error function $E(w)$ as the limit of the sequence $\{E[w(\tau)]\colon \tau = 0, 1, 2, ...\}$, where $w(0)$ is an initial estimate of the local minimizer, say w^*. They generate a sequence of points $\{w(\tau)\colon \tau = 0, 1, 2, ...\}$ which can be thought of as defining a discrete approximation from the initial point $w(0)$ to the minimum.

The four local minimization procedures (i)-(iv) above have different properties. The gradient descent procedure has the advantage of being rather simple and cheap to implement. But it may show extremely slow convergence in some cases. The Newton procedure, in contrast, has the advantage of rapid convergence when the starting point $w(0)$ is sufficiently close to the minimizer w^*. The procedure, however, is relatively expensive to implement since it requires the evaluation of the inverse Hessian matrix (second-order derivatives) of the error function.

The quasi-Newton and conjugate gradient procedures are intrinsically off-line parameter adjustment techniques, and evidently more sophisticated local optimization procedures. The quasi-Newton requires the evaluation and storage in memory of a totally dense matrix $H(\tau)$ at each iteration step. Thus, for large Q the storage requirements are extremely large. The conjugate gradient methods require much less storage than the quasi-Newton procedure. But they require an exact determination of the learning rate $\eta(\tau)$ and the parameter $\beta(\tau)$ in each learning step τ. In addition, they need approximately twice as many gradient evaluations as the quasi-Newton procedure (Cichocki and Unbehauen 1994, p. 91).

5 Batch versus on-line optimization

It is important to note that error functions based on maximum likelihood for a set of N independent observations comprise a sum of terms, one for each data point, so that

$$E(w) = \sum_{n=1}^{N} E_n\left[w(x_n, t_n)\right] \tag{33}$$

where $E_n(w)$ is called the local and $E(w)$ the global error function. There are two basic approaches to find the minimum of the global error function $E(w)$. The first approach is called *batch optimization*. In this approach the total error function is minimized in such a way that the parameter changes are accumulated over all training examples before the

parameters are actually changed. Minimization procedures that use the whole data set at once are called *batch* procedures.

Local minimization procedures used for batch optimization might work well, but have difficulties when the error surface is flat (that is, gradients close to zero), when gradients can be in a very large range, or when the surface is very rugged. When gradients can vary greatly, the search may progress too slowly when the gradient is small, and may overshoot when the gradient is large. When the error surface is rugged, a local search from a random starting point usually converges to a local minimum and a worse situation than the global minimum (Shang and Wah 1996, p. 46).

The second approach to find the minimum of the global error function is the *on-line* [also known as *pattern-based*] optimization. On-line optimization makes an update to the parameter vector based on one data point at a time. This update is repeated by cycling through the training data set either in sequence or by selecting points at random with replacement.

On-line compared to batch based optimization has several advantages. One advantage is that on-line procedures introduce some randomness (noise) that often may help in escaping from local minima, since a stationary point with respect to the error function for the entire data set will generally not be a stationary point for each data point individually. Second, on-line optimization is usually more convenient than batch optimization when the number of the training examples is very large, since batch optimization requires auxiliary memory to accommodate the local updates. Third, on-line procedures are usually faster and more effective than standard batch procedures, especially for large-scale classification problems. This may be explained by the fact that many training examples may posses redundant information in the sense that many

contributions to the gradient are very similar, and waiting to compute all these contributions before updating the parameters is simply wasteful. On the other hand, if high precision is required, batch optimization may be the approach of choice (Cichocki and Unbehauen 1994, pp. 135-136).

When the surface modelled by the error function is extremely rugged and has many local minima, then a local search from a random starting point leads to converge to a local minimum close to the initial point. In order to find a sufficiently good minimum, it may be necessary to run a minimization procedure multiple times, each time using a different randomly chosen starting point $w(0)$, and comparing the resulting performance on an independent validation data set.

Alternatively, stochastic global search procedures might be used. Examples of such procedures include Alopex (see, for example, Fischer 2002, Fischer et al. 2003), genetic algorithms (see, for example, Fischer and Leung 1998), and simulated annealing. These procedures guarantee convergence to a global solution with a higher probability, but at the expense of slower convergence.

Finally, it is worth noting that local minimization procedures in batch or on-line mode need not only a starting point, but also a stopping rule. The stopping rule does not need a form of central control. Many *ad hoc* stopping rules had been proposed. One which appears to be popular is to have a validation data set, and stop training when the error function on the validation data set starts to rise. But one can never know if the minimum error on the validation set has yet been attained (Ripley 1996, pp. 154-155).

6 The technique of error backpropagation

This section describes an efficient technique for evaluating the gradient of an error function $E(w)$ for a feedforward neural network model that is required for network training. This technique – sometimes simply termed *backprop* – uses a local message passing scheme in which information is sent alternately forward and backward through the network. Its modern form stems from Rumelhart et al. (1986) illustrated for on-line gradient descent optimization applied to the sum-of-squares error function. It is important to recognize, however, that the technique 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 on-line or batch mode.

We describe the backpropagation technique for a general neural network of type (6) that has two parameter layers, arbitrary differentiable activation functions $\phi(\bullet)$ and $\psi(\bullet)$ with a corresponding local error function E_n. The resulting backpropagation formulae will then be illustrated using a network structure that has logistic sigmoid hidden and linear output units associated with the simple sum-squared error function.

For each pattern *n* in the training data set, we shall assume that we have supplied the corresponding input vector $x_n = (x_{n1}, ..., x_{nM})'$ to the network, and calculated the activations of all the hidden and output units in the network by applying Eqs. (1) and (3). Recall that each hidden unit has input a_{hn} and output $z_{hn} = \phi(a_{hn})$, and each output unit k has input a_{kn} and output $y_{kn} = \psi(a_{kn})$. This process is called *forward propagation*, because it can be seen as a forward flow of information provided by x_n

through the network. For the rest of this section we consider one training example and drop the subscript *n* in order to keep the notation uncluttered.

Recall from Eq. (33) that the error functions which we consider (those defined by maximum likelihood for a set of *iid* data) can be written as a sum of an error for each pattern *n* separately so that

$$E(w) = \sum_{n=1}^{N} E_n(w) . \tag{34}$$

We consider the problem of evaluating $\nabla E_n(w)$ for one of the local error functions. This may be used then for on-line optimization, or the results can be accumulated over the whole training data set in the case of batch optimization.

We evaluate the gradient of E_n with respect to a hidden-to-output parameter $w_{kh}^{(2)}$ first, by noting that E_n depends on the weights $w_{kh}^{(2)}$ $(k = 1, ..., K; h = 1, ..., H)$ only via the summed input, a_k [see Eq. (3)] to the output unit *k*. Thus, we can apply the chain rule for partial derivatives to get

$$\frac{\partial E_n}{\partial w_{kh}^{(2)}} = \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial w_{kh}^{(2)}} . \tag{35}$$

We now introduce a useful notation

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k} \tag{36}$$

where the δs are often referred to as errors. Using Eq. (3), we can write

$$\frac{\partial a_k}{\partial w_{kh}^{(2)}} = z_h . \tag{37}$$

Substituting Eqs. (36) and (37) into Eq. (35), we then get

$$\frac{\partial E_n}{\partial w_{kh}^{(2)}} = \delta_k \, z_h . \tag{38}$$

This equation tells us that the required partial derivative with respect to $w_{kh}^{(2)}$ is obtained simply by the multiplication of two terms: the value of δ for unit *k* at the output end and the value of *z* at the input end *h* of the connection concerned, where $z = 1$ in the case of a bias node.

We note that from the definition (36) we have

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k} = \psi'(a_k) \, \frac{\partial E_n}{\partial y_k} \tag{39}$$

where $\psi(\bullet)$ is the activation function of the output units, and the use of the prime signifies differentiation with respect to the argument. In order to evaluate (39) we substitute appropriate expressions for $\psi'(a_k)$ and $\partial E_n / \partial y_k$.

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 deltas are given by

$$\delta_k = y_k - t_k \tag{40}$$

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

$$\delta_k = y_k(1-y_k)(y_k - t_k). \tag{41}$$

Let us move to evaluate the gradient of E_n with respect to an input-to-hidden parameter $w_{hm}^{(1)}$, which is more deeply embedded in the error function. Using again the chain rule for partial derivatives, we get

$$\frac{\partial E_n}{\partial w_{hm}^{(1)}} = \delta_h \frac{\partial a_h}{\partial w_{hm}^{(1)}} = \delta_h x_m \tag{42}$$

where we have defined

$$\delta_h \equiv \phi'(a_h)\sum_{k=1}^{K} \delta_k w_{kh}^{(2)}. \tag{43}$$

In the case of a logistic sigmoid activation function ϕ we get the following backpropagation formula

$$\delta_h = \phi'(a_h)\sum_{k=1}^{K}\delta_k\, w_{kh}^{(2)} = \phi(a_h)\,[1-\phi(a_h)]\sum_{k=1}^{K}\delta_k\, w_{kh}^{(2)} = z_h(1-z_h)\sum_{k=1}^{K}\delta_k\, w_{kh}^{(2)}. \qquad (44)$$

Since the formula for δ_h $(h=1,...,H)$ contains only terms in a later layer, it is clear that it can be calculated from output to input on the network. Hence 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 through the network, followed by a backward pass to calculate δ_k and δ_h, and thus 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 x_n to the network and forward propagate through the network, using Eqs. (1) and (3) to generate the hidden and output activations a_h and a_k.

Step 2: Evaluate the δ_k for all the *K* output units using Eq. (40) or Eq. (41), depending on the problem to be studied.

Step 3: Backpropagate the deltas, using Eq. (43) to get δ_h for each hidden unit *h* in the network model.

Step 4: Use Eqs. (38) and (42) to evaluate the required derivatives.

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

To illustrate the technique of error backpropagation let us consider a two-layer network of the form illustrated in Fig. 1, together with a sum-of-squares error function. The output units have linear activation functions while the hidden units logistic sigmoid activation functions ϕ given by Eq. (7). A useful property of this function is that its derivative can be expressed in a particularly simple form as $\phi'(a_h) = \phi(a_h)[1-\phi(a_h)]$.

For the sum-of-squares error function *E* the error for pattern *n* is given by

$$E_n = \frac{1}{2}\sum_{k=1}^{K}\left(y_k - t_k\right)^2 \tag{45}$$

where y_k is the response of output unit *k*, and t_k is the corresponding target, for a particular input pattern x_n. For each pattern in the training set in turn, we first perform a forward propagation using

$$a_h = \sum_{m=0}^{M} w_{hm}^{(1)} x_m \tag{46}$$

$$z_h = \phi(a_h) = \frac{1}{1+\exp(-a_h)} \tag{47}$$

$$y_k = \sum_{h=0}^{H} w_{kh}^{(2)} z_h \,. \tag{48}$$

Next, we compute the deltas for each output unit using Eq. (40) since the hidden units have logistic sigmoid activation functions

$$\delta_k = y_k - t_k \tag{40}$$

and then propagate these to obtain the deltas for the hidden units using

$$\delta_h = z_h(1 - z_h)\sum_{k=1}^{K} w_{kh}^{(2)}\,\delta_k \tag{49}$$

where the sum runs over all output units. The derivatives with respect to the first layer and second layer weights are then given by

$$\frac{\partial E_n}{\partial w_{hm}^{(1)}} = \delta_h\, x_m \tag{50}$$

$$\frac{\partial E_n}{\partial w_{kh}^{(2)}} = \delta_k\, z_h\,. \tag{51}$$

Note that Eq. (51) has the same form as Eq. (50), but with a different definition of the deltas.

Using fixed step on-line gradient descent optimization, for example, where η is constant and fixed in the training process, the weights in the first and second layers are updated using

$$\Delta w_{hm}^{(1)} = -\eta\, \delta_h\, x_m \tag{52}$$

$$\Delta w_{kh}^{(2)} = -\eta\, \delta_k\, z_h\,. \tag{53}$$

In the case of batch gradient descent optimization, the weights would be updated according to

$$\Delta w_{hm}^{(1)} = -\eta \sum_{n=1}^{N} \delta_{hn}\, x_{mn} \tag{54}$$

$$\Delta w_{kh}^{(2)} = -\eta \sum_{n=1}^{N} \delta_{kn}\, z_{hn}\,. \tag{55}$$

It is worth noting that the technique of error backpropagation can also be applied to the calculation of second order derivatives of the error function required for example for the application of Newton's procedure (see Bishop 2006, pp. 249-250; Ripley 1996, pp. 151-153).

The fact that the error function derivatives can be computed by backpropagating errors is clearly attractive. The update rules (52) and (53) are local. To compute the parameter change for a given connection one only needs quantities available (after backpropagation of the deltas) at the ends of that connection. This makes the backpropagation technique appropriate for parallel computation. The technique is also computationally efficient. If we have Q connections in all, computation of the error function takes the order Q operations, so calculating Q derivatives directly would take

order Q^2 operations. In contrast the backpropagation technique lets us calculate all the derivatives in order of Q operations (Hertz et al. 1991, p. 119).

7 Optimizing complexity in the model selection process

So far we have considered feedforward neural network models of type (6) with a priori given numbers of input, hidden and output units. While the number of input and output units is essentially problem dependent, the number H of hidden units is a free parameter that can be adjusted to provide the best predictive performance. Hence one might expect that in a maximum likelihood setting there will be an optimal value of H that gives the best predictive performance corresponding to the optimum between overfitting and underfitting (Bishop 2006, p. 256).

A network model that is too simple (i.e., H too small), or too inflexible, will have a large bias and smooth out some of the underlying structure in the data (corresponding to a high bias), while one that has too much flexibility (i.e., H is too large) 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 will be poor. This highlights the need to optimize the complexity in the model selection process, in order to achieve the best predictive performance (Bishop 1995, p. 332, Fischer 2000).

Choosing the number of hidden units is a difficult issue, since the generalization error is not a simple function of H and this is due to the presence of local minima in the error function. One approach to selecting H is to train a sequence of neural network

models with an increasing number of hidden units and then to select that one which yields the best predictive performance on a testing data set.

There are, however, more principled ways to control the complexity of a neural network model. One approach is that of *regularization* (Poggio and Giroso 1990; Bishop 1995, pp. 338-353). The idea behind this approach is to define a criterion to select an approximate solution from a set of admissible solutions. The basic feature of regularization is a compromise between the fidelity to data and fidelity to some prior information about the solution. In other words, the regularization approach imposes a weak smoothness constraint on the possible solution (Cichocki and Unbehauen 1994, pp. 248-250).

According to this approach one starts with a relatively large value for H and then controls complexity by adding a regularization (also called penalty or complexity) term to the error function in order to discourage the parameters from reaching large values. The regularized error function (that is, the functional to be minimized) $\tilde{E}$ is the weighted sum of two terms

$$\tilde{E}(w, \mu) = E(w) + \mu R(w) \tag{56}$$

where $E(w)$ is one of the error functions as discussed in the section on 'network training and the error functions', and $R(w)$ is the smoothness constraint. This term embodies a prior knowledge about the solution, and thus depends on the nature of the particular problem at hand. The regularization parameter μ (a positive real number) controls the smoothness of degree of fit of the regularized solution. In practice, μ is usually chosen by trail and error.

The simplest such regularization term takes the form of a sum of all the parameters, giving a modified error function

$$\tilde{E}(w,\mu) = E(w) + \mu \frac{1}{2} \|w\|_2^2 \tag{57}$$

where $\|w\|_2^2 = w'w,$ and w is the Q-dimensional parameter vector. Note that often the biases are omitted from the regularizer or it may be included but with its own regularization coefficient. The case of a quadratic regularizer is called ridge regression in the statistics literature, and known as weight decay in the context of neural networks. The effective model complexity is determined by the choice of $\mu.$

Sometimes, a generalized version of Eq. (57) is used, for which the regularized error function takes the form

$$\tilde{E}(w,\mu) = E(w) + \mu \|w\|_m^m \tag{58}$$

where $\|.\|_m$ denotes the L_m-norm. Note that the case $m = 2$ corresponds to the quadratic regularizer given by Eq. (57). The case $m = 1$ is known as the 'lasso' in the statistics literature (Tibshirani 1996). The regularizer given by Eq. (58) has the property that – if μ is sufficiently large – some of the parameters are driven to zero in on-line optimization, 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, however, is its inconsistency with certain scaling characteristics of network mappings (see Bishop 1995, pp. 340-342). If, for example, one trains a network model using 'original' data and trains the same model using data from which the input variables are linearly transformed, then consistency requires that the regularizer should be invariant to re-scaling of the weights and to shifts of the biases (Bishop 2006, p. 258).

A regularized which is invariant under linear transformations in the above sense is given by

$$\mu_1 \|w_1\|_m^m + \mu_2 \|w_2\|_m^m \tag{59}$$

where w_1 denotes the Q_1-dimensional vector of weights $(Q_1 = HM)$ in the first layer and w_2 the Q_2-dimensional vector of parameters $(Q_2 = KH)$ in the second layer. This regularizer will remain unchanged under linear transformations of the weights, provided that the regularization coefficients μ_1 and μ_2 are suitably rescaled.

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, hence, shifted to one of determining a suitable value for the regularization coefficients during training.

The principal alternative to regularization as an approach to optimizing the effective complexity of a neural network model is *early stopping* (also known as stopped training). As we have seen in the previous sections, network training corresponds to an iterative reduction of the error function defined with respect to a given training data set. For many of the local minimization procedures used for network

training (such as, for example, gradient descent), the error is a non-increasing function of the iteration steps. But the error measured with respect to independent data (generally known as *validation data set*) often shows a decrease first, followed by an increase as the network model starts to overfit.

Network training can hence be stopped at the point of smallest error with respect to the validation data, in order to get a network that shows good predictive performance. If the validation data set, however, is small, it may be necessary to keep aside another data set (known as the test data set), on which the performance of the network model is finally evaluated. For an application of this approach in the context of spatial interaction modelling see Fischer and Gopal (1994).

This approach of stopping training before a minimum of the training error has been reached is another way of controlling the effective complexity of a network. 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 parameter 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, for example, 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 approach. In particular, it has been shown that stopped training has the same sort of regularization effect (that is reducing model variance at the cost of bias) that regularization terms provide.

8 Generalization performance

Feedforward neural networks can be seen as flexible non-linear models for regression and classification, that are especially useful in data-rich, but theory poor application contexts. Failures in applications can generally be attributed to inadequate network training (indicated by the presence of overfitting or underfitting), and inadequate complexity of the network (in other words, inappropriate size of the hidden layer).

The real test of how a neural network model performs in a specific context essentially relates to its ability to give good predictions for new data. The simplest way to assess generalization performance of a neural network model is to use a test set independent of the training set (and validation set if used for early stopping), and to evaluate the model's error by means of the chosen error function. As the training and test sets are independent samples, an unbiased estimate of the prediction error can be obtained. But this approach becomes practical only if the data sets are very large or new data can be generated cheaply.

One way to overcome the problem of data scarcity is by *cross-validation*. Cross-validation is a sample re-use procedure for assessing generalization performance. It makes maximally efficient use of the available data. The idea is to divide the available data set U_N into – generally equally sized – parts, and then use one part to test the performance of the neural network model trained on the remaining parts. The resulting estimator is again unbiased.

With small samples of data – that is, when structural uncertainty is greatest – cross-validation may not be feasible, because there are too few data values with which to perform estimation and testing in a stable way. Bootstrapping the neural network

modelling process – that is creating bootstrap copies of the available data to generate copies of training and testing sets – may be used instead as a general framework for evaluating generalization performance. For introductory text books on the bootstrapping approach see Efron (1982), Efron and Tibshirani (1993), and Hastie et al. (2001), and for application in the context of spatial interaction modelling Fischer (2002), Fischer and Reismann (2002a, b).

9 Closing remarks

There is no doubt in mind that there are many regression and classification problems in the social sciences in general and in economic geography in particular, in which non-linear models such as neural network models can outperform classical (that is, linear) models, and as automated data collection increases such problems will continue to proliferate.

It has been commonplace to view neural networks as kinds of black boxes, and this leads to inappropriate applications which may fail not because such models cannot work but because the issues of model specification, estimation and generalization performance are not well understood. Failures in applications can usually be attributed to inadequate learning and/or inadequate complexity of the network model. Parameter estimation and a suitably chosen number of hidden units are, thus, of crucial importance for the success of real world applications. The chapter view network learning as an optimization problem, describes various learning procedures, provides insights into

current best practice to optimize complexity and suggests the use of the bootstrap pairs approach to evaluate the model's generalization performance.

The value of this chapter is, thus, primarily in providing an understanding of the feedforward neural networks and their behaviour so that one can design an appropriate network model for an appropriate regression or classification problem.

Bibliography

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

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

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

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

Chen K., Leung Y., Leung K.-S. and X. Gao (2002), ,A neural network for solving nonlinear programming problems', *Neural Computing Problems*, **11** (2), 103-111.

Cichocki A. and R. Unbehauen (1994), *Neural Networks for Optimization and Signal Processing*. Chichester: John Wiley.

Cybenko G. (1989), 'Approximation by superpositions of a sigmoidal function', *Mathematics of Control Signals and Systems*, **2** (4), 303-314.

Dang C., Leung, Y., Gao X. and K. Chen (2004), 'Neural networks for nonlinear and mixed complementarity problems and their applications', *Neural Networks*, **17** (2), 271-283.

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

Efron B. and R. Tibshirani (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], Piscataway [NJ]: IEEE Press, pp. 1880-1882.

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

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

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

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

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

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

Fletcher R. (1986), *Practical Methods for Optimization*, New York: Wiley-Interscience.

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

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

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

Leung Y. (1994), 'Inference with spatial knowledge. An artificial neural network approach', *Geographical Systems*, **1** (2), 103-112.

Leung Y., Gao X. B. and K. Z. Chen (2004), 'A dual neural network for solving entropy-maximizing models', *Environment and Planning* A, **36** (5), 879-919.

Leung Y., Zhang J. and Z. Xu (1997), 'Neural Networks for convex hull computation', *IEEE Transactions on Neural Networks*, **8** (3), 601-611.

Luenberger P. (1984), *Linear and Nonlinear Programming*, Reading [MA]: Addison-Wesley.

Mozolin A., Thill J.-C. and E.L. Usery (2000), 'Trip distribution with multilayer perceptron neural networks: a critical evaluation', *Transportation Research* B, **34** (1), 53-73.

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

Patuelli R., Reggiani A., Nijkamp P. and N. Schaune (2011), 'Neural networks for regional employment forecasts: Are the parameters relevant?' *Journal of Geographical Systems*, **13** (1), 67-85.

Poggio T. and F. Giroso (1990), 'Networks for approximation and learning', *Proceedings of the IEEE* 78, pp. 1481-1497.

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

Reggiani A. and T. Tritapepe (1998). 'Neural networks and logit models applied to commuters' mobility in the metropolitan area of Milan', in V. Himanen, P. Nijkamp and A. Reggiani (eds), *Neural Networks in Transport Applications*, Aldershot: Ashgate, pp. 111-129.

Ripley B. D. (1996), *Pattern Recognition and Neural Networks*, 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.

Rumelhart D. E., Hinton G. E. 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*, Cambridge [MA]: MIT Press, pp. 318-362.

Tibshirani R. (1996), 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society B*, **58** (1), 267-288.

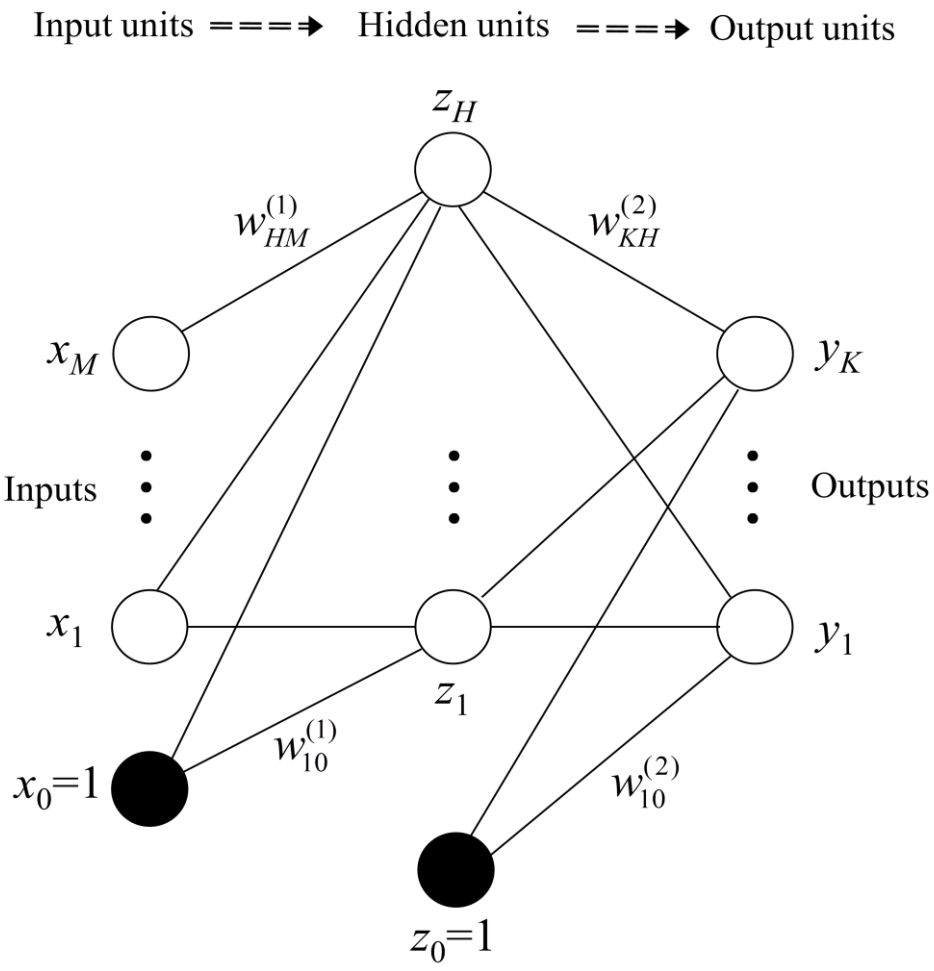
Shang Y. and B. W. Wah (1996), 'Global optimization for neural network training', *Computer*, **29** (3), 45-54.

Stinchcombe M. and H. White (1989), 'Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions', *Proceedings of the International Joint Conference on Neural Networks* I, pp. 612-617.

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

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

Figure 1


Input units ====➔ Hidden units ====➔ Output units
z_H
$w_{HM}^{(1)}$
$w_{KH}^{(2)}$
x_M
y_K
Inputs
Outputs
x_1
y_1
z_1
$w_{10}^{(1)}$
$w_{10}^{(2)}$
$x_0=1$
$z_0=1$