C1.2 Multilayer perceptrons

Luis B Almeida

Abstract

This section introduces multilayer perceptrons, which are the most commonly used type of neural network. The popular backpropagation training algorithm is studied in detail. The momentum and adaptive step size techniques, which are used for accelerated training, are discussed. Other acceleration techniques are briefly referenced. Several implementation issues are then examined. The issue of generalization is studied next. Several measures to improve network generalization are discussed, including cross validation, choice of network size, network pruning, constructive algorithms and regularization. Recurrent networks are then studied, both in the fixed point mode, with the recurrent backpropagation algorithm, and in the sequential mode, with the unfolding in time algorithm. A reference is also made to time-delay neural networks. The section also includes brief mention of a large number of applications of multilayer perceptrons, with pointers to the bibliography.

C1.2.1 Introduction

Multilayer perceptrons (MLPs) are the best known and most widely used kind of neural network. They are formed by units of the type shown in figure C1.2.1. Each of these units forms a weighted sum of its inputs, to which a constant term is added. This sum is then passed through a nonlinearity, which is often called its *activation function*. Most often, units are interconnected in a *feedforward manner*, that is, with interconnections that do not form any loops, as shown in figure C1.2.2. For some kinds of applications, recurrent (i.e. nonfeedforward) networks, in which some of the interconnections form loops, are also used. B3.2.4

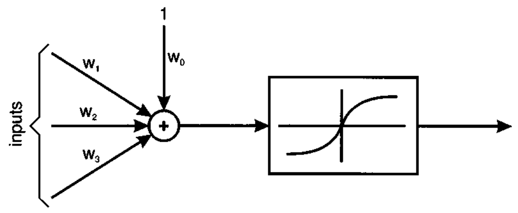

Figure C1.2.1. A unit of a multilayer perceptron.

Training of these networks is normally performed in a supervised manner. One assumes that a *training set* is available, which contains both input patterns and the corresponding desired output patterns (also called *target patterns*). As we shall see, the training is normally based on the minimization of some error measure between the network's outputs and the desired outputs. It involves a backward propagation through a network similar to the one being trained. For this reason the training algorithm is normally called *backpropagation*.

In this chapter we will study multilayer perceptrons and the backpropagation training algorithm. We will review some of the most important variants of this algorithm, designed both for improving the training speed and for dealing with different kinds of networks (feedforward and recurrent). We will also briefly

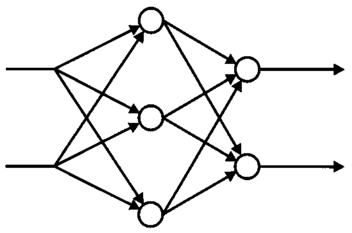

Figure C1.2.2. Example of a feedforward network. Each circle represents a unit of the type indicated in figure C1.2.1. Each connection between units has a weight. Each unit also has a bias input, not depicted in this figure.

mention some theoretical and practical issues related to the use of multilayer perceptrons and other kinds of supervised networks.

C1.2.2 Network architectures

We saw in figure C1.2.2 an example of a feedforward network, of the type that we will consider in this chapter. As we noted above, the interconnections of the units of this network do not form any loops, and hence the network is said to be *feedforward*. Networks in which there are one or more loops of interconnections, such as the one in figure C1.2.3, are called *recurrent*. B2.3

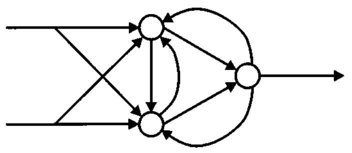

Figure C1.2.3. A recurrent network.

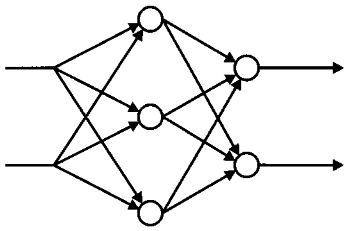

Figure C1.2.4. A layered network.

In feedforward networks, units are often arranged in layers, as in figure C1.2.4, but other topologies can also be used. Figure C1.2.5 shows a network type that is useful in some applications, in which direct links between inputs and output units are used. Figure C1.2.6 shows a three-unit network that is fully connected, i.e. that has all the interconnections that are allowed by the feedforward restriction.

The nonlinearities in the network's units can be any differentiable functions, as we shall see below. The kind of nonlinearity that is most commonly used has the general form shown in figure C1.2.7. It has two horizontal asymptotes, and is monotonically increasing, with a single point where the curvature changes sign. Curves with this general shape are usually called *sigmoids*. Some of the most common expressions of sigmoids are B3.2.4

$$S(s) = \frac{1}{1+\mathrm{e}^{-s}} = \frac{1+\tanh(s/2)}{2} \tag{C1.2.1}$$

$$S(s) = \tanh(s) \tag{C1.2.2}$$

$$S(s) = \arctan(s)\,. \tag{C1.2.3}$$

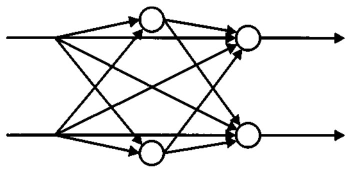

Figure C1.2.5. A network with direct links between input and output units.

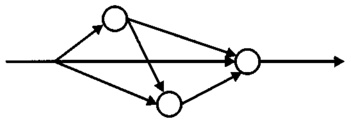

Figure C1.2.6. A fully connected feedforward network.

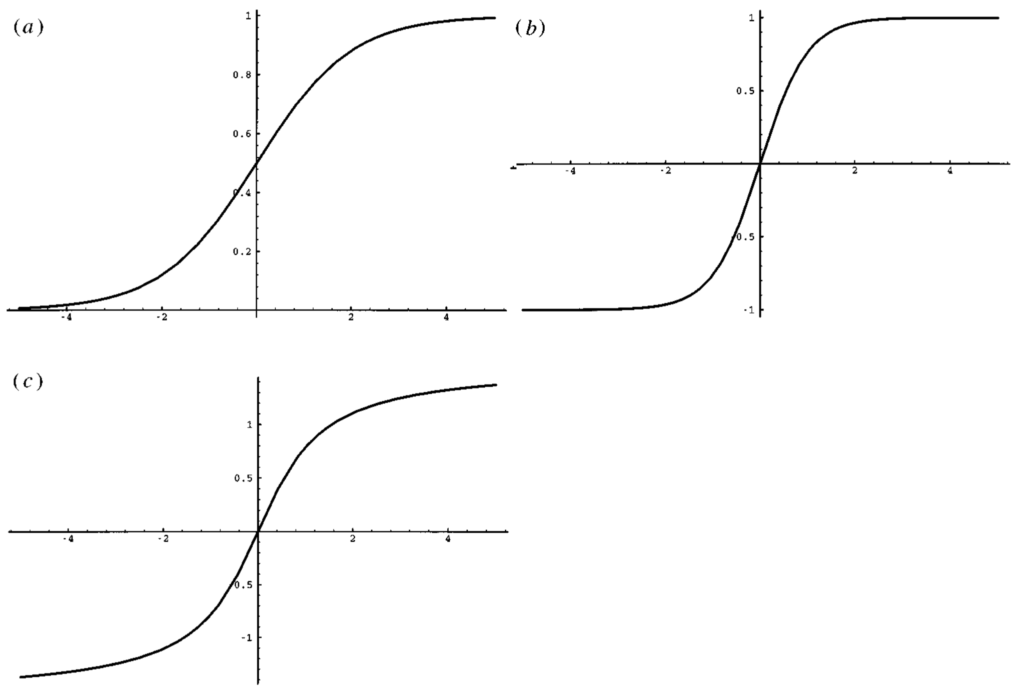

Figure C1.2.7. Sigmoids corresponding to: (*a*) equation (C1.2.1), (*b*) equation (C1.2.2) and (*c*) equation (C1.2.3).

Sigmoid (C1.2.3) is sometimes scaled to vary between -1 and $+1$. Sigmoid (C1.2.1) is often designated as the *logistic function*. As we said above, interconnections between units have *weights*, that multiply the values which go through them. Besides the variable inputs that come through weighted links, units normally also have a fixed input, which is often called *bias*.

It is through the variation of the weights and biases that networks are trained to perform the operations that are desired from them. As an example of how weight changes can affect the behavior of networks, figure C1.2.8 shows three one-unit networks that differ in their weights and that perform different logical operations. Figure C1.2.9 shows two networks with different topologies, that both perform the logical XOR operation. These two networks were trained by the backpropagation algorithm, to be described below. Note that since these networks have analog outputs, the output values are often not exactly 0 or 1. A usual convention, for binary applications, is that output values above the middle of the range of the sigmoid are taken as *true* or 1, and output values below that are taken as *false* or 0. This is the convention adopted here.

As we shall see below, it is sometimes convenient to consider input nodes as units of a special kind, which simply copy the input components to their outputs. These units are then normally designated as *input units*. The number of units and the number of layers that a given network is said to have may depend on whether this convention is taken or not. Another convention that is normally made is to designate as *hidden units* the units that are internal to the network, i.e. those units that are neither input nor output units. The two networks of figure C1.2.9 have, respectively, two and one hidden units.

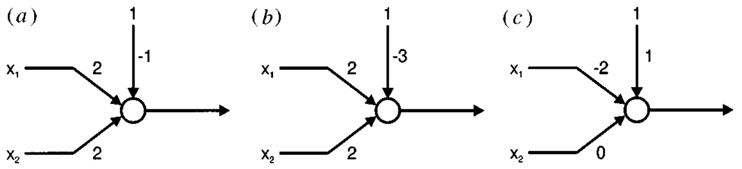

Figure C1.2.8. Single-unit networks implementing simple Boolean functions. (*a*) OR. (*b*) AND. (*c*) NOT. The units are assumed to have logistic nonlinearities.

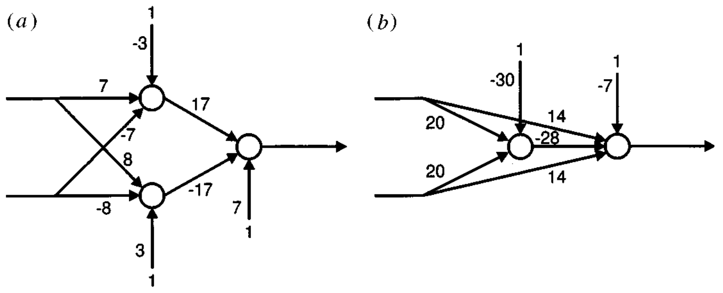

Figure C1.2.9. Two networks that have been trained to perform the XOR operation. The units are assumed to have logistic nonlinearities. The weight values have been rounded, for convenience.

C1.2.3 The backpropagation algorithm for feedforward networks

Let us represent the input pattern of a network by an m-dimensional vector $\boldsymbol{x}$ (italic bold characters shall represent vectors) and the outputs of the units of the network by an N-dimensional vector $\boldsymbol{y}$. To keep the notation compact, we will represent the input nodes of the network as units (numbered from 1 to m). These units simply copy the components of the input pattern, i.e.

$$y_i = x_i \qquad i = 1, \ldots, m\,.$$

We will also assume that there is a unit number 0, whose output is fixed at 1, i.e. $y_0 = 1$. The weights from this unit to other units of the network will represent the bias terms of those units. The remaining units, $m+1$ to N, are the operative units, that have the form shown in figure C1.2.1. In this way, all the parameters of the network appear as weights in interconnections among units, and can therefore be treated jointly, in a common manner. Denoting by w_{ji} the weight in the branch that links unit j to unit i, we can write the weighted sum performed by unit i as

$$s_i = \sum_{j=0}^{N} w_{ji} y_j \qquad i = m+1, \ldots, N\,. \tag{C1.2.4}$$

Note that w_{0i} represents the unit's bias term and w_{ji}, with $j = 1, \ldots, m$, are the weights linking the inputs to unit i. We will make the convention that if a branch from one unit to another does not exist in the network, the corresponding weight is set to zero. The unit's output will be

$$y_i = S(s_i) \qquad i = m+1, \ldots, N \tag{C1.2.5}$$

where S represents the unit's nonlinearity. For the sake of simplicity, we shall assume that the same nonlinearity is used in all units of the network (it would be straightforward to extend the reasoning in this chapter to situations in which nonlinearities differ from one unit to another). As we shall see, the only restriction on the nonlinearities is that they must be differentiable. The output pattern of the network is formed by the outputs of one or more of its units. We will collect these outputs into the output vector $\boldsymbol{o}$.

Let us denote by $\boldsymbol{x}^k$ the kth pattern of the training set. We assume the training set to have K patterns (the training sets that are most often used are of finite size; infinite-sized training sets are sometimes used, and this would imply slight modifications in what follows, essentially amounting to changing the sums over training patterns into series or integrals, as appropriate). If we assume that we are presenting $\boldsymbol{x}^k$ at the input of the network, we can define an error vector $\boldsymbol{e}^k$ between the actual outputs $\boldsymbol{o}^k$ and the desired outputs $\boldsymbol{d}^k$ for the current input pattern:

$$\boldsymbol{e}^k = \boldsymbol{o}^k - \boldsymbol{d}^k \, . \tag{C1.2.6}$$

The squared norm of the error vector, $E^k = \|\boldsymbol{e}^k\|^2$ can be seen as a scalar measure of the deviation of the network from its ideal behavior, for the input pattern $\boldsymbol{x}^k$. In fact, E^k is zero if $\boldsymbol{o}^k = \boldsymbol{d}^k$. Otherwise it is positive, progressively increasing as the network outputs deviate from the desired ones. We can define a measure of the network's deviation from the ideal, in the whole training set, as

$$E = \sum_{k=1}^{K} E^k \tag{C1.2.7}$$

where K is the number of patterns of the training set. If the training set and the network architecture are fixed, E is only a function of the weights of the network, that is, $E = E(\boldsymbol{w})$ (when convenient, we will assume that we have collected all the weights as components of a single vector $\boldsymbol{w}$). We can think of the task of training the network on the given training set as the task of finding the weights that minimize E. If there is a set of weights that yields $E = 0$, then a successful minimization will result in a network that performs without error in the whole training set. Otherwise, the weights that minimize E will correspond to the network that performs best in the quadratic error sense.

The quadratic error may not be the best measure of the deviation from ideal in all situations, though it is by far the most commonly used one. If convenient, however, some other cost function $C(\boldsymbol{e})$ can be used, with $E^k = C(\boldsymbol{e}^k)$. The total cost to be minimized is still given by (C1.2.7). The cost function C should be chosen so as to represent, as closely as possible, the relative importances of different errors in the situation where the network is to be applied. In general, $C(\boldsymbol{e})$ has an absolute minimum for $\boldsymbol{e} = \boldsymbol{0}$, and in what follows the only restriction on C is that it be differentiable relative to all components of $\boldsymbol{e}$.

C1.2.3.1 The basic algorithm

There are, in the mathematical literature, several different methods for minimizing a function such as $E(\boldsymbol{w})$. Among these, one that results in a particularly simple procedure is the gradient method. Essentially, this method consists of iteratively taking steps, in weight space, proportional to the negative gradient of the function to be minimized, that is, of iteratively updating the weights according to

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n - \eta \boldsymbol{\nabla} E \tag{C1.2.8}$$

where $\boldsymbol{\nabla} E$ represents the gradient of E relative to $\boldsymbol{w}$. This iteration is repeated until some appropriate stopping criterion is met. If $E(\boldsymbol{w})$ obeys some mild regularity conditions and η is small enough, this iteration will converge to a local minimum of E. The parameter η is normally designated as the *learning rate parameter* or *step size parameter*.

The main issue in applying this algorithm is the computation of the gradient components, $\partial E/\partial w_{ji}$. For feedforward networks, this computation takes a very simple form (Bryson and Ho 1969, Werbos 1974, Parker 1985, Le Cun 1985, Rumelhart *et al* 1986). This is best described by means of an example. Consider the network of figure C1.2.10(a). From this network we obtain another one (figure C1.2.10(b)) as follows: we first linearize all nonlinear elements of the original network, replacing them by linear branches with gains $g_i = S'(s_i)$. We then *transpose* it (Oppenheim and Schafer 1975) that is, we reverse the direction of flow of all branches, replacing summing nodes by divergence nodes and vice-versa, and changing outputs into inputs and vice-versa. This new network is often called the *backpropagation network*, or *error propagation network*, for reasons that will soon become clear. As indicated in the figure, we denote the variables in this network by the same letters as the corresponding ones in the MLP, with an overbar.

For feedforward networks, the *backpropagation rule* for computing the gradient components, which we shall describe next, can be easily derived by repeated application of the chain rule of differentiation;

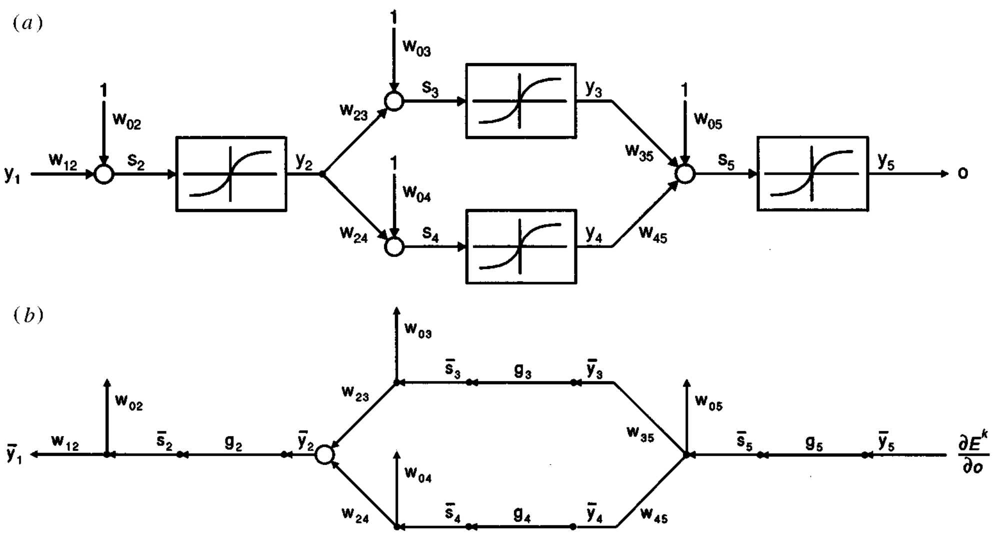

Figure C1.2.10. Example of a multilayer perceptron and of the corresponding backpropagation network. (*a*) Multilayer perceptron. (*b*) Backpropagation network, also called error propagation network.

see for example (Rumelhart *et al* 1986). We will not make that derivation here, however, because in section C1.2.8.1 we will make the derivation for a certain class of recurrent networks that includes feedforward networks as a special case. Here, we will therefore simply describe the rule. First of all, note that, from (C1.2.7)

$$\frac{\partial E}{\partial w_{ji}} = \sum_k \frac{\partial E^k}{\partial w_{ji}} .$$

We place the pattern $\boldsymbol{x}^k$ at the inputs of the MLP, we compute the output error according to (C1.2.6) and we place at the inputs of the error propagation network the values $\partial E^k/\partial o_i$ as shown in figure C1.2.10. The backpropagation rule states that the partial derivatives can then be obtained as

$$\frac{\partial E^k}{\partial w_{ji}} = y_j \bar{s}_i \tag{C1.2.9}$$

i.e. the partial derivative relative to a weight is the product of the inputs of the branches corresponding to that weight in the MLP and in the backpropagation network. As we said, the proof of this fact will be given in section C1.2.8.1.

If the quadratic error is used as a cost function, then $\partial E^k/\partial o_i = 2e_i^k$. Since the backpropagation network is linear, we can place at its inputs e_i^k, instead of $2e_i^k$, and compute the derivatives according to

$$\frac{\partial E^k}{\partial w_{ji}} = 2 y_j \bar{s}_i . \tag{C1.2.10}$$

In this case the backpropagation network is propagating errors. This justifies the name of *error propagation network* that is commonly given to the backpropagation network. The variables $\bar{s}_i$ are often called *propagated errors*.

To apply this training procedure, we must have a training set, containing a collection of input patterns and the corresponding target outputs, and we must select a network architecture to be trained (number of units, arranged or not in layers, interconnections among units, activation functions). We must also choose an initial weight vector, $\boldsymbol{w}^1$ (weights are normally initialized in a random manner, usually with a uniform distribution in some symmetric interval $[-a, a]$—see section C1.2.5.3 below), a step size parameter η and an appropriate stopping criterion.

The backpropagation algorithm can be summarized as follows, where we denote by K the number of patterns in the training set.

(i) Set $n = 1$. Repeat steps (a) through (c) below until the stopping criterion is met.

  (a) Set the variables g_{ji} to zero. These variables will be used to accumulate the gradient components.

(b) For $k = 1, \ldots, K$ perform steps (1) through (4).
   (1) Propagate forward: apply the training pattern $\boldsymbol{x}^k$ to the perceptron and compute its internal variables y_i and outputs $\boldsymbol{o}^k$.
   (2) Compute the cost function derivatives: compute $\partial E^k/\partial o_i^k$.
   (3) Propagate backwards: apply $\partial E^k/\partial o_i^k$ to the inputs of the backpropagation network and compute its internal variables $\bar{s}_i$.
   (4) Compute and accumulate the gradient components: compute the values $\partial E^k/\partial w_{ji} = y_j \bar{s}_i$ and accumulate each of them in the corresponding variable, i.e. $g_{ji} = g_{ji} + y_j \bar{s}_i$.

(c) Update the weights: set $w_{ji}^{n+1} = w_{ji}^n - \eta g_{ji}$. Increment n.

This algorithm can be used with any differentiable cost function. When the quadratic error is used as a cost function, the factor 2 that appears in (C1.2.10) is usually incorporated into the learning rate constant η, and steps (2) to (4) are replaced by the following.

   (2) Compute the output errors: compute $\boldsymbol{e}^k = \boldsymbol{o}^k - \boldsymbol{d}^k$.
   (3) Propagate backwards: apply e_i^k to the inputs of the backpropagation network and compute its internal variables $\bar{s}_i$.
   (4) Compute and accumulate the gradient components: compute the values $y_j \bar{s}_i$ and accumulate each of them in the corresponding variable, $g_{ji} = g_{ji} + y_j \bar{s}_i$.

For finite minima, i.e. for minima that are not situated at infinity, the above algorithm is guaranteed to converge for η below a certain value $\eta_{\max}$, if the activation functions and the cost function are continuous and differentiable. However, the upper bound $\eta_{\max}$ depends on the network, on the training set and on the cost function, and cannot be specified in advance. On the other hand, the fastest convergence is normally obtained for an optimal value of η that is somewhat below this upper bound. For η below the optimal value, the convergence speed can decrease considerably. This makes the choice of the learning rate parameter η a critical aspect of the training procedure. Often, preliminary tests have to be made with different learning rates, in order to try to find a good value of η for the problem to be solved. In section C1.2.4.2 we will describe a modification of the algorithm, involving adaptive step sizes, which solves this difficulty almost completely, and also yields faster training.

The stopping criterion to be used depends on the problem being addressed. In some situations, the training is stopped when the cost function E becomes lower than some prescribed value. In other situations, the algorithm is stopped when the maximum absolute value of the error components e_i^k becomes lower than some given limit. In other situations still, training is stopped when the variation of E or of the weights becomes too slow. Often, an upper bound on the number of iterations n is also incorporated, to prevent the algorithm from running forever if the chosen conditions are never met.

C1.2.3.2 Stochastic backpropagation

When the training set is large, each weight update (which involves a sweep through the whole training set) may become very time-consuming, making learning very slow. In such cases, another version of the algorithm, performing a weight update per pattern presentation, can be used.

(i) Set $n = 1$. Repeat step (a) below until the stopping criterion is met.
   (a) For $k = 1, \ldots, K$, perform steps (1) through (5).
      (1) Propagate forward: apply the training pattern $\boldsymbol{x}^k$ to the perceptron, and compute its internal variables y_i and outputs $\boldsymbol{o}^k$.
      (2) Compute the cost function derivatives: compute $\partial E^k/\partial o_i^k$.
      (3) Propagate backwards: apply $\partial E^k/\partial o_i^k$ to the inputs of the backpropagation network, and compute its internal variables $\bar{s}_i$.
      (4) Compute the gradient components: compute the values $\partial E^k/\partial w_{ji} = y_j \bar{s}_i$.
      (5) Update the weights: set $w_{ji}^{n+1} = w_{ji}^n - \eta y_j \bar{s}_i$. Increment n.

To differentiate between the two forms of the algorithm, the former is often qualified as *batch*, *off-line* or *deterministic*, while the latter is called *real-time*, *on-line* or *stochastic*. This last designation stems from the fact that, under certain conditions, the latter form of the algorithm implements a *stochastic gradient descent*. Its convergence can then be guaranteed if η is varied with n, in such a way that (i) $\eta(n) \to 0$ and

(ii) $\sum_{n=1}^{\infty} \eta(n) = \infty$. In fact, the algorithm can then be shown to satisfy the conditions for convergence introduced by Ljung (1978). In practice, since any training is in fact finite, it is not always clear how best to decrease η. A solution that is sometimes used is to train first in real-time mode, until convergence becomes slow, and then switch to batch mode. Frequently, the largest speed advantage of real-time training occurs in the first part of the training process, and the later switch to batch mode does not bring about any significant increase in training time.

Backpropagation is a generalization of the delta rule for training single linear units: *adalines*. In fact, C1.1.3 it is easy to see that, when applied to a single linear unit (i.e. a unit without nonlinearity), backpropagation coincides with the delta rule. For this reason, backpropagation is sometimes designated the *generalized* B3.2.4 *delta rule*.

C1.2.3.3 Local minima

An issue that may have already come to the reader's mind is that gradient descent, like any other local optimization algorithm, converges to local minima of the function being minimized. Only by chance will it converge to the global minimum. A solution that can be used to try to alleviate this problem is to perform several independent trainings, with different random initializations of the weights. Even this, however, does not guarantee that the global minimum will be found, although it increases the probability of finding lower local minima. On the other hand, this solution cannot be used for large problems, where training times of days or even weeks can be involved. When the function $E(\boldsymbol{w})$ is very complex, with many local minima, one must essentially abandon the hope of finding the optimum, and accept local minima as the best that can be found. If these are good enough, the problem is solved. Otherwise, the only viable solution normally involves using a more complex architecture (e.g. with more hidden units, and/or with more layers) that will normally have lower local minima. It must be said, however, that although local minima are a drawback in the training of multilayer perceptrons, they do not usually cause too many difficulties in practice.

C1.2.3.4 Universal approximation property

An important property of feedforward multilayer perceptrons is their universality, that is, their capacity to approximate, to any desired accuracy, any desired function. The main result in this respect was first obtained by Cybenko (1989), and later, independently, by Funahashi (1989) and by Hornik *et al* (1989). It shows that a perceptron with a single hidden layer of sigmoidal units and with a linear output unit can uniformly approximate any continuous function in any hypercube (and therefore also in any closed, bounded set). More specifically, it states that, if a function f, continuous in a closed hypercube $H \subset \mathbb{R}^k$, and an error bound $\varepsilon > 0$ are given, then a number h, weight vectors $\boldsymbol{w}_i$ and output weights a_i $(i = 1, \ldots, h)$ exist such that the output of the single hidden layer perceptron

$$o(\boldsymbol{x}) = \sum_{i=1}^{h} a_i S(\boldsymbol{w}_i \cdot \boldsymbol{x})$$

approximates f in H with an error smaller than ε, that is, $|f(\boldsymbol{x}) - o(\boldsymbol{x})| < \varepsilon$ for all $\boldsymbol{x} \in H$, if the nonlinearity S is continuous, monotonically increasing and bounded. Here, for compactness of notation, we have assumed that the input vector $\boldsymbol{x}$ has been extended with a component $x_0 = 1$ and that the weight vectors $\boldsymbol{w}_i$ have components from 0 to k, so that the inner product $(\boldsymbol{w}_i \cdot \boldsymbol{x})$ incorporates a bias term.

This result is rather reassuring, since it guarantees that even perceptrons with a single hidden layer can approximate essentially all useful functions. However, the limitations of this result should also be understood. First of all, the theorem only guarantees the existence of a network, but does not provide any constructive method to find it. Second, it does not give any bounds on the number of hidden units h needed for approximating a given function to a desired level of accuracy. It may well turn out that, for some specific problems, while a single hidden layer perceptron must exist which gives a good enough approximation to the desired result, either it is too hard to find, or it has too large a number of hidden units (or both). A large number of units, and therefore of weights, may be a strong drawback, meaning that a very large number of training patterns is required for adequately training the network (see the discussion on generalization in section C1.2.6). On the other hand, it may happen that networks with more than one hidden layer can yield the desired approximation with a much smaller number of weights. The situation is somewhat similar to what happens with combinatorial digital circuits. Although any digital function can be implemented in two layers (e.g. by expressing it as a sum of products), a complex function, such as an output of a binary adder for a large word size, can require an intractable number of product terms, and therefore of gates in the first layer. However, by using more layers, the implementation may become easily tractable.

C1.2.4 Accelerated training

The training of multilayer perceptrons by the backpropagation algorithm is often rather slow, and may require thousands or tens of thousands of *epochs*, in complex problems (the name *epoch* is normally given to a training sweep through the whole training set, either in batch or in real-time mode). The essential reason for this is that the error surface, as a function of the weights, normally has narrow ravines (regions where the curvature along one direction is rather strong, while it is very weak in an orthogonal direction, the gradient component along the latter direction being very small). In these regions, the use of a large learning rate parameter η will lead to a divergent oscillation across the ravine. A small η will lead the weight vector to the 'bottom' of the ravine, and convergence to the minimum will then proceed along this bottom, but at a very low speed, because the gradient and η are both small. In the next sections we will describe two methods of *improving the training speed* of multilayer perceptrons, especially in situations B3.4 where narrow ravines exist.

C1.2.4.1 Momentum technique

Let us rewrite the weight update equation C1.2.8 as

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n + \Delta\boldsymbol{w}^n$$

with

$$\Delta\boldsymbol{w}^n = -\eta\boldsymbol{\nabla}E\,.$$

The momentum technique (Rumelhart *et al* 1986) replaces the latter equation with

$$\Delta\boldsymbol{w}^n = -\eta\boldsymbol{\nabla}E + \alpha\boldsymbol{w}^n$$

in which $0 \le \alpha < 1$. The second term in the equation, called the *momentum term*, introduces a kind of B6.3.3 'inertia' in the movement of the weight vector, since it makes successive weight updates similar to one another, and has an accumulation effect, if successive gradients are in similar directions. This increases the movement speed along the ravine, and helps to prevent oscillations across it. This effect can also be seen as a linear low-pass filtering of the gradient $\boldsymbol{\nabla}E$. The effect becomes more pronounced as α approaches 1, but normally one has to be conservative in the choice of α because of an adverse effect of the momentum term: the ravines are normally curved, and in a bend the weight movement may be up a ravine wall, if too much momentum has been previously acquired. Like the learning rate parameter η, the momentum parameter α has to be appropriately selected for each problem. Typical values of α are in the range 0.5 to 0.95. Values below 0.5 normally introduce little improvement relative to backpropagation without momentum, while values above 0.95 often tend to cause divergence at bends. The momentum technique may be used both in batch and real-time training modes. In the latter case, the low-pass filtering action also tends to smooth the randomness of the gradients computed for individual patterns.

With momentum, the batch-mode backpropagation algorithm becomes the following.

(i) Set $n = 1$ and $\Delta w_{ji}^0 = 0$. Repeat steps (a) through (d) below until the stopping criterion is met.
  - (a) Set the variables g_{ji} to zero. These variables will be used to accumulate the gradient components.
  - (b) For $k = 1, \ldots, K$ (where K is the number of training patterns), perform steps (1) through (4).
    - (1) Propagate forward: apply the training pattern $\boldsymbol{x}^k$ to the perceptron and compute its internal variables y_j and outputs $\boldsymbol{o}^k$.
    - (2) Compute the cost function derivatives: compute $\partial E^k/\partial o_i^k$.
    - (3) Propagate backwards: apply $\partial E^k/\partial o_i^k$ to the inputs of the backpropagation network and compute its internal variables $\bar{s}_i$.
    - (4) Compute and accumulate the gradient components: compute the values $\partial E^k/\partial w_{ji} = y_j\bar{s}_i$ and accumulate each of them in the corresponding variable, i.e. $g_{ji} = g_{ji} + y_j\bar{s}_i$.

(c) Apply momentum: set $\Delta w_{ji}^{n} = -\eta g_{ji} + \alpha \Delta w_{ji}^{n-1}$

(d) Update the weights: set $w_{ji}^{n+1} = w_{ji}^{n} + \Delta w_{ji}^{n}$. Increment n.

The real-time backpropagation algorithm with momentum is

(i) Set $n = 1$ and $\Delta w_{ji}^{0} = 0$. Repeat step (a) below until the stopping criterion is met.

- (a) For $k = 1, \ldots, K$, perform steps (1) through (6).
  - (1) Propagate forward: apply the training pattern $\boldsymbol{x}^k$ to the perceptron and compute its internal variables y_j and outputs $\boldsymbol{o}^k$.
  - (2) Compute the cost function derivatives: compute $\partial E^k / \partial o_i^k$.
  - (3) Propagate backwards: apply $\partial E^k / \partial o_i^k$ to the inputs of the backpropagation network and compute its internal variables $\bar{s}_i$.
  - (4) Compute the gradient components: compute the values $\partial E^k / \partial w_{ji} = y_j \bar{s}_i$.
  - (5) Apply momentum: set $\Delta w_{ji}^{n} = -\eta y_j \bar{s}_i + \alpha \Delta w_{ji}^{n-1}$.
  - (6) Update the weights: set $w_{ji}^{n+1} = w_{ji}^{n} + \Delta w_{ji}^{n}$. Increment n.

C1.2.4.2 Adaptive step sizes

The adaptive step size method is a simple acceleration technique, proposed in Silva and Almeida (1990a, b) for dealing with ravines. For related techniques see Jacobs (1988) and Tollenaere (1990). It consists of using an individual step size parameter η_{ji} for each weight, and adapting these parameters in each iteration, depending on the successive signs of the gradient components:

$$\eta_{ji}^{n} = \begin{cases} \eta_{ji}^{n-1} u & \text{if } \left(\frac{\partial E}{\partial w_{ji}}\right)^{n} \text{ and } \left(\frac{\partial E}{\partial w_{ji}}\right)^{n-1} \text{ have the same sign} \\ \eta_{ji}^{n-1} d & \text{if } \left(\frac{\partial E}{\partial w_{ji}}\right)^{n} \text{ and } \left(\frac{\partial E}{\partial w_{ji}}\right)^{n-1} \text{ have different signs} \end{cases} \qquad \text{(C1.2.11)}$$

$$\Delta w_{ji}^{n} = -\eta_{ji}^{n} \frac{\partial E}{\partial w_{ji}} \qquad \text{(C1.2.12)}$$

where $u > 1$ and $d < 1$. There are two basic ideas behind this procedure. The first is that, in ravines that are parallel to some axis, use of appropriate individual step sizes is equivalent to eliminating the ravine, as discussed in Silva and Almeida (1990b). Ravines that are not parallel to any axis but are not too diagonal either, are not completely eliminated, but are made much less pronounced. The second idea is that quasi-optimal step sizes can be found by a simple strategy: if two successive updates of a given weight were performed in the same direction, then its step size should be increased. On the other hand, if two successive updates were in opposite directions, then the step size should be decreased.

As is apparent from the explanation above, the adaptive step size technique is especially useful for ravines that are parallel, or almost parallel, to some axis. Since the technique is less effective for ravines that are oblique to all axes, use of a combination of adaptive step sizes and the momentum term technique is justified. This combination is normally done by replacing (C1.2.12) with

$$z_{ji}^{n} = \frac{\partial E}{\partial w_{ji}} + \alpha z_{ji}^{n-1}$$
$$\Delta w_{ji}^{n} = -\eta_{ji}^{n} z_{ji}^{n}$$

that is, we first filter the gradient with the momentum technique, and then multiply the filtered momentum by the adaptive step sizes.

For applying the backpropagation algorithm with adaptive step sizes and momentum, one must choose the following parameters:

η_0	initial value of the step size parameters
u	‘up’ step size multiplier
d	‘down’ step size multiplier
α	momentum parameter.

Typical values, which will work well in most situations, are $u = 1.2$, $d = 0.8$ and $\alpha = 0.9$. The initial value of the step size parameters is not critical, but is normally chosen small to prevent the algorithm from diverging in the initial epochs, while the step size adaptation still did not have enough time to act. The step size parameters will then be increased by the step size adaptation algorithm, if necessary. If the robustness measures indicated in section C1.2.4.3 are incorporated in the algorithm, even large initial step size parameters will not cause divergence, and essentially any value can be chosen for η_0.

The batch-mode training algorithm with adaptive step sizes and momentum is as follows.

(i) Set $n = 1$, $\eta_{ji}^1 = \eta_0$ and $z_{ji}^0 = 0$. Repeat steps (a) through (d) below until the stopping criterion is met.

- (a) Set the variables g_{ji}^n to zero. These variables will be used to accumulate the gradient components.
- (b) For $k = 1, \ldots, K$ (where K is the number of training patterns), perform steps (1) through (4).
  - (1) Propagate forward: apply the training pattern $\boldsymbol{x}^k$ to the perceptron and compute its internal variables y_j and outputs $\boldsymbol{o}^k$.
  - (2) Compute the cost function derivatives: compute $\partial E^k / \partial o_i^k$.
  - (3) Propagate backwards: apply $\partial E^k / \partial o_i^k$ to the inputs of the backpropagation network and compute its internal variables $\bar{s}_i$.
  - (4) Compute and accumulate the gradient components: compute the values $\partial E^k / \partial o_i^k$ and accumulate each of them in the corresponding variable, i.e. $g_{ji}^n = g_{ji}^n + y_j \bar{s}_i$.
- (c) Apply momentum: set $z_{ji}^n = g_{ji}^n + \alpha z_{ji}^{n-1}$.
- (d) Adapt the step sizes: if $n \geq 2$ set

$$\eta_{ji}^n = \begin{cases} u\eta_{ji}^{n-1} & \text{if } g_{ji}^n \text{ and } g_{ji}^{n-1} \text{ have the same sign} \\ d\eta_{ji}^{n-1} & \text{if } g_{ji}^n \text{ and } g_{ji}^{n-1} \text{ have opposite signs}. \end{cases}$$

- (e) Update the weights: set $w_{ji}^{n+1} = w_{ji}^n - \eta_{ji}^n z_{ji}^n$. Increment n.

The adaptive step size technique was designed, in principle, for batch training. It has, however, been used with success in real-time training, with the following modifications: (i) while weights are adapted after every pattern presentation, step sizes are adapted only at the end of each epoch, and (ii) instead of comparing the signs of the derivatives, in the step size adaptation (C1.2.11), we compare the signs of the total changes of the weight in the last and next to last epochs.

C1.2.4.3 Robustness

As was said in section C1.2.3.1, the step size parameter η has to be small enough for the backpropagation algorithm to converge. During the course of training, either with or without adaptive step sizes, one may come to a region of weight space for which the current step size parameters are too large, causing an increase in the cost function from one epoch to the next. A similar increase can also occur in a curved ravine if too much momentum has previously been acquired, as noted in section C1.2.4.1. To prevent the cost function from increasing, one must then go back to the step with lowest cost function, reduce the step size parameters and set the momentum memory to zero. To do this, after each epoch we must compare the current value of the cost function with the lowest that was ever found in the current training, and take the above-mentioned measures if the current value is higher than that lowest one (a small tolerance for cost function increases is allowed, as we will see below). To be more specific, these measures are as follows.

(i) Return to the set of weights that produced the lowest value of the cost function.
(ii) Reduce all the step size parameters (or the single step size parameter, if adaptive step sizes are not being used) by multiplying by a fixed factor $r < 1$.
(iii) Set the momentum memories z_{ji}^{n-1} (or Δw_{ji}^{n-1} if adaptive step sizes are not being used) to zero.

After this, an epoch is again executed. If the error still increases, the same measures are repeated: returning to the previous point, reducing step sizes and setting momentum memories to zero. This repetition continues until an error decrease is observed. The normal learning procedure is then resumed. A value that is often used for the reduction factor is $r = 0.5$. A tolerance is normally used in the comparison of values of the cost function, that is, a small increase is allowed without taking the measures indicated above. In batch mode, the allowed increase is very small (e.g. 0.1%) just to allow for small numerical errors in the computation of the cost function. In real-time mode, a larger increase (e.g. 20%) has to be allowed, because the exact cost function is normally never computed. Instead, the cost function contributions from the different patterns are added during a whole epoch, while the weights are also being updated. This sum of cost function contributions is only an estimate of the actual cost function at the end of the epoch, and this is why a larger tolerance is needed. If desired, the actual cost function could be computed at the end of each epoch, by presenting all the patterns while keeping the weights frozen, but this would increase computation significantly.

The procedure described in this section is rather effective in making the training robust, irrespective of whether it is combined with adaptive step sizes and/or momentum or not. When combined with adaptive step sizes and momentum, it yields a very effective MLP training algorithm.

C1.2.4.4 Other acceleration techniques

In this section we will summarize other existing techniques for *fast MLP training*. Most of them are based on a local second-order approximation to the cost function, attempting to reach the minimum of that approximation in each step (for a review of a number of variants see Battiti (1992)). These techniques make use of the Hessian matrix, that is, of the matrix of second derivatives of the cost function relative to the weights. Some methods compute the full Hessian matrix. Since the number of elements of the Hessian is the square of the number of weights, these methods have the important drawback that their amount of computation per epoch is proportional to that square. These methods reduce the number of training epochs but, for large networks, they involve a very large amount of computation per epoch. Other methods assume that the Hessian is diagonal, thereby achieving a linear growth of the computation per epoch with the number of weights. Among these, a variant (Becker and Le Cun 1989) estimates the diagonal elements of the Hessian through a backward propagation, similar to the one described in section C1.2.3.1 for computing the gradient. Another variant, called *quickprop* (Fahlman 1989) estimates the second derivatives based on the variation of the first derivatives from one epoch to the next. It should be noted that the adaptive step size algorithm described in section C1.2.4.2, and the related algorithms referenced in that section, can also be viewed as indirect ways to estimate diagonal Hessian elements. B3.4

Another class of second-order techniques is based on the method of conjugate gradients (Press *et al* 1986). This is a method which, when employed with a second-order function, can find its minimum in a number of steps equal to the number of arguments of the function. The various conjugate gradient techniques that are in use differ from one another, essentially, in the approximations they make to deal with non-second-order functions. Among these techniques, one of the most effective appears to be the one of Moller (1990).

We should not conclude this section without mentioning that, when the input patterns have few components (up to about 5–10), networks of local units (e.g. *radial basis function networks*) are normally much faster to train than multilayer perceptrons. However, as the dimensionality of the input grows, networks of local units tend to require an exponentially large number of units, making their training very long, and requiring very large training sets to be able to generalize well (cf section C1.2.6). B1.7.3, C1.6.2

C1.2.5 Implementation

In this section we discuss some issues that are related to the practical implementation of multilayer perceptrons and of the backpropagation algorithm.

C1.2.5.1 Sigmoids

As we said above, the activation functions that are most commonly used in units of multilayer perceptrons are of the sigmoidal type. Other kinds of nonlinearities have sometimes been tried, but their behavior generally seems to be inferior to that of sigmoids. Within the class of sigmoids there still is, however, a wide room for choice. The characteristic of sigmoids that appears to have the strongest influence on the performance of the training algorithm is symmetry relative to the origin. Functions like the hyperbolic tangent and the arctangent are symmetric relative to the origin, while the logistic function, for example, is symmetric relative to a point of coordinates $(0, 0.5)$. Symmetry relative to the origin gives sigmoids a bipolar character that normally tends to yield better conditioned error surfaces. Sigmoids like the logistic tend to originate narrow ravines in the error function, which impair the speed of the training procedure (Le Cun *et al* 1991).

C1.2.5.2 Output units and target values

Most practical applications of multilayer perceptrons can be divided, in a relatively clear way, into two different classes. In one of the classes, the target outputs take a continuous range of values, and the task of the network is to perform a nonlinear regression operation. Normally, in this case, it is convenient not to place nonlinearities in the outputs of the network. In fact, we normally wish the outputs to be able to span the whole range of possible target values, which is often wider than the range of values of the sigmoids. We could, of course, scale the amplitudes of the output sigmoids appropriately, but this rarely has any advantage relative to the simple use of units without nonlinearities at the outputs. Output units are then said to be linear. They simply output the weighted sum of their inputs plus their bias term.

In the other class, which includes most classification and pattern recognition applications, the target outputs are binary, that is, they take only two values. In this case it is common to use output units with sigmoid nonlinearities, similar to other units in the network. The binary target values that are most appropriate depend on the sigmoids that are used. Often, target values are chosen equal to the two asymptotic values of the sigmoids (e.g. 0 and 1 for the logistic function, and ± 1 for the tanh and the scaled arctan functions). In this case, to achieve zero error, the output units would have to achieve full saturation, i.e. their input sums would have to become infinite. This fact would tend to drive the weights linking to these units to grow indefinitely in absolute value, and would slow down the training process. To improve training speed, it is therefore common to use target values that are close, but not equal, to the asymptotic values of the sigmoids (e.g. 0.05 and 0.95 for the logistic function, and ± 0.9 for the tanh and the scaled arctan functions).

C1.2.5.3 Weight initialization

Before the backpropagation algorithm can be started, it is necessary to set the weights of the network to some initial values. A natural choice would be to initialize them all with a value of zero, so as not to bias the result of training in any special direction. However, it can easily be seen, by applying the backpropagation rule, that if initial weights are zero, all gradient components are zero (except for those that concern weights on direct links between input and output units, if such links exist in the network). Moreover, those gradient components will always remain at zero during training, even if direct links do exist. Therefore, it is normally necessary to initialize the weights to nonzero values. The most common procedure is to initialize them to random values, drawn from a uniform distribution in some symmetric interval $[-a, a]$. As we mentioned above, several independent trainings with independent random initializations may be used, to try to find better minima of the cost function.

It is easy to understand that large weights (resulting from large values of a) will tend to saturate the respective units. In saturation the derivative of the sigmoidal nonlinearity is very small. Since this derivative acts as a multiplying factor in the backpropagation, derivatives relative to the unit's input weights will be very small. The unit will be almost 'stuck', making learning very slow.

If the inputs to a given unit i in the network all have similar root mean square (rms) values and are all independent from one another, and if the weights are initialized in some given, fixed interval, the rms value of the unit's input sum will be proportional to $(f_i)^{1/2}$, where f_i is the number of inputs of unit i (often called the unit's *fan-in*). To keep the rms values of the input sums similar to one another, and to avoid saturating the units with largest fan-ins, the parameter a, controlling the width of the initialization interval, is sometimes varied from unit to unit, by making $a_i = k/(f_i)^{1/2}$. There are different preferences for the choice of k. Some people prefer to initialize the weights very close to the origin, making k very small (e.g. 0.01 to 0.1), and therefore keeping the units in their central linear regions in the beginning of the training process. Other people prefer larger values of k (e.g. 1 or larger), that lead the units into their nonlinear regions right from the start of training.

C1.2.5.4 Input normalization and decorrelation

Let us consider the simplest network that one can design, formed by a single linear unit. Single-unit linear networks (adalines) have been in use for a long time, in the area of discrete-time signal processing.

Finite-impulse response (FIR) filters (Oppenheim and Schafer 1975) can actually be viewed as single linear units with no bias. The inputs are consecutive samples of the input signal, and the weights are the filter coefficients. Therefore, adaptive filtering with FIR filters is essentially a form of real-time training of linear-unit networks. It is therefore no surprise that the first adaptive filtering algorithms were derived from the delta rule (Widrow and Stearns 1985).

It is a well-known fact from adaptive filter theory that training is fastest, because the error function is best conditioned (without any ravines) if the inputs to the linear unit are uncorrelated among themselves, that is, $\langle x_i x_j \rangle = 0$ for $i \neq j$, and have equal mean-squared values, that is, $\langle x_i^2 \rangle = \langle x_j^2 \rangle$ for all i, j. Here $\langle\cdot\rangle$ represents the expected value (most often, when training perceptrons, the expected value can be estimated simply by averaging in the training set).

If a bias term is also used in the linear unit, it acts as an extra input that is constantly equal to 1. Its mean squared value is 1, and therefore the mean squared values of all other inputs should also be equal to 1. On the other hand, cross-correlations of other inputs with this new input are simply the expected values of those other inputs, which should be equal to zero, as all cross-correlations between inputs: $\langle x_i 1 \rangle = \langle x_i \rangle = 0$. In summary, for fastest training of a single linear unit with bias one should preprocess the data so that the average of each input component is zero,

$$\langle x_i \rangle = 0$$

and the components are decorrelated and normalized:

$$\langle x_i x_j \rangle = \delta_{ij}$$

where δ_{ij} is the Kronecker symbol. It has been found by experience that this kind of preprocessing also tends to accelerate the training in the case of multilayer perceptrons. Setting the averages of input components to zero can simply be performed by adding an appropriate constant to each of them. Decorrelation can then be performed by any orthogonalization procedure, for example, the Gram–Schmidt technique (Golub and Van Loan 1983). Finally, normalization can be performed by an appropriate scaling of each component. The most cumbersome of these steps is the orthogonalization, and people sometimes skip it, simply setting means to zero and mean-squared values to one. This simplified preprocessing is usually designated *input normalization*, and is often quite effective at increasing the training speed of networks. A more elaborate acceleration technique, involving the adaptive decorrelation and normalization of the inputs of all layers of the network, is described in (Silva and Almeida 1991).

C1.2.5.5 Shared weights

In some cases one would wish to constrain some weights of a network to be equal to one another. This situation may arise, for example, if we wish to perform the same kind of processing in various parts of the input pattern. It is a common situation in image processing, where one may want to detect the same feature in different parts of the input image. An example, in a handwritten digit application, is given in (Le Cun *et al* 1990a). Two examples of shared weight situations will also be found below, in the discussion of recurrent networks.

The difficulty in handling shared weights comes from the fact that even if these weights are initialized with the same value, the derivatives of the cost function relative to each of them will usually be different from one another. The solution is rather simple. Assume that we have collected all weights in a weight vector $\boldsymbol{w} = (w_1, w_2, \ldots)^{\mathrm{T}}$ (where T denotes transposition), and that the first m weights are to be kept equal to one another. These weights are not, in fact, free arguments of the cost function E. To keep all of the arguments of E free, one should replace all of these weights by a single argument a, to which all of them will be equal. Then, the partial derivative of E should be computed relative to a, and not relative to each of these weights individually. But

$$\begin{aligned}\frac{\partial E}{\partial a} &= \sum_{i=1}^{m} \frac{\partial E}{\partial w_i}\frac{\partial w_i}{\partial a} \\ &= \sum_{i=1}^{m} \frac{\partial E}{\partial w_i}\,.\end{aligned}$$

The derivatives that appear in the last line can be computed by the normal backpropagation procedure. In summary, one should compute the derivatives relative to each of the individual weights in the normal way, and then use their sum to update a and therefore to update all the shared weights. One should also remember that shared weights should be initialized to the same value.

C1.2.6 Generalization

Until now we have been discussing the training of multilayer perceptrons based on the assumption that we wish to optimize their performance (measured by the cost function) in the training set. However, this is a simplification of the situation that we normally find in practice. Consider, for example, a network being trained to perform a classification task. We assume that we are given a training set, which is usually finite, containing examples of the desired classification. This set is usually only a minute fraction of the universe in which the network will be used after training. After training, the network will be used to classify patterns that were not in the training set.

We see that ideally we would like to minimize the cost function computed in the whole universe. That is normally either impossible or impractical, however, because the universe is infinite, because we do not know it all in advance, or simply because that would be too costly in computational terms. Until now we have been using the cost function evaluated in the training set as an estimate of its value in the whole universe. Whenever possible, precautions should be taken to ensure that the training set is as representative of the whole universe as possible. This may be achieved, for example, by randomly drawing patterns from the universe, to form the training set. Even if this is done, however, the statistical distribution of the training set will only be an approximation to the distribution of the universe. A consequence of this is that, since we optimize the performance of the network in the training set, its performance in that set will normally be better than in the whole universe. A network whose performance in the universe is similar to the performance in the training set is said to *generalize* well, while a network whose performance degrades B3.5 significantly from the training set to the universe is said to generalize poorly.

These facts have two main implications. The first is that if we wish to have an unbiased estimate of the network's performance in the universe, we should not use the performance in the training set, but rather in a *test set* that is independent from the training set. The second implication is that we should try to design networks and training algorithms in order to ensure good generalization, and not only good performance in the training set.

C1.2.6.1 Network size

An important issue in what concerns generalization is the size of the network. Intuitively, it is clear that one cannot effectively train a large network with a training set containing only a few patterns. Consider a network with a single output. When we present at the input a given training pattern, we can idealize writing an expression of the output of the network as a function of the weights. If we wish to make the output equal to the desired output, we can set that expression equal to the desired output, and we will obtain an equation whose unknowns are the weights. The whole training set will therefore yield a set of equations. If the network has more than one output, the situation is similar, and the number of equations will be the number of training patterns times the number of outputs. These equations are usually nonlinear and very complex, and therefore not solvable by conventional means. They may even have no exact solution. Training algorithms are methods to find exact or approximate solutions for such sets of equations.

By making an analogy with the well-known case of the systems of linear equations, we can gain some insight into the issue of generalization. If the number of unknowns (i.e. weights) is larger than the number of equations, there will generally be an infinite number of solutions. Since each of these solutions corresponds to a different set of weights, it is clear that they will generalize differently from one another, and only by chance will the specific solution that we find generalize well. If the number of weights is equal to the number of equations, a linear system will usually have a single solution. A nonlinear system will usually have no solutions, a single solution or a finite number of solutions. Since these are optimal for the training set, which is different from the universe, they will still often not generalize well. The interesting situation is the one in which there are fewer weights than equations. In this case, there will be no solution, unless the set of equations is redundant. Even the existence of an approximate solution implies that there must be some kind of redundancy, or regularity, in the training set (e.g. in a digit-recognition problem, regularities are the facts that all zeros have a round shape, all ones are approximately vertical bars, and so on). With fewer weights than training patterns, the only way for the network to approximately satisfy the training equations is to exploit the regularities of the problem, and the fewer weights the network has, the more it will have to rely on the training set's regularities to be able to perform well on that set. But these regularities are exactly what we expect to be maintained, from the training set to the universe. *Therefore, small networks, with fewer weights than the number of equations, are the ones that can be expected to generalize best, if they can be trained to perform well on the training set*. Note that the latter condition means that network topology is a very important factor. A network with the appropriate number of weights but with an inappropriate topology will not be able to perform well in the training set, and therefore cannot also be expected to perform well in the universe. On the other hand, a network with an appropriately small number of weights and with the appropriate topology will be able to perform well in the training set, and also to generalize well. As a rule of thumb, we would say that the number of weights should be around or below one tenth of the product of the number of training patterns by the number of outputs. In some situations, however, it may go up to about one half of that product.

There are other methods to try to improve generalization. The methods that we will mention are *stopped training*, *network pruning*, *constructive techniques* and the use of a *regularization term*.

C1.2.6.2 Stopped training and cross-validation

In *stopped training*, one considers all the successive weight vectors found during the course of the training process, and tries to find the vector that corresponds to the best generalization. This is normally done by *cross-validation*. Another set of patterns, independent from the training and test sets, is used to evaluate the network's performance during the training (this set of patterns is often designated the *validation set*). At the end of training, instead of selecting the weights that perform best in the training set, we select the weights that performed best in the validation set. This is equivalent, in fact, to performing an early stop of the training process, before convergence in the training set, which justifies the designation of 'stopped training'. Since the performance in the validation set tends to oscillate significantly during the training process, it is advisable to continue training even after the first local minimum in the validation performance is observed, because better validation performance may still arise later in the process. Note that, since the validation set is used to select the set of weights to be kept, it effectively becomes part of the training data, i.e. the performance of the final network in the validation set is not an unbiased estimate of its performance on the universe. Therefore, an independent test set is still required, to evaluate the network's performance after training is complete. B3.5.2 B3.5.2

C1.2.6.3 Pruning and constructive techniques

Network pruning techniques start from a large network, and try to successively eliminate the least important interconnections, thereby arriving at a smaller network whose topology is appropriate for the problem at hand, and which has a good probability of generalizing well. Among the pruning techniques we mention the skeletonization method of Mozer and Smolensky (1989), *optimal brain damage* (Le Cun *et al* 1990b) and *optimal brain surgeon* (Hassibi *et al* 1993). Network pruning, while effective, tends to be rather time-consuming, since after each pruning some retraining of the network has to be performed (an interesting and efficient technique, which is a blend of pruning and regularization, is mentioned below in section C1.2.6.4). Constructive techniques work in the opposite way to pruning: they start with a small network and add units until the performance is good enough. Several constructive techniques have appeared in the literature, the best known of which is probably cascade-correlation (Fahlman and Lebiere 1990). Other constructive techniques can be found in Frean (1990) and Mézard and Nadal (1989). B3.5.2

C1.2.6.4 Regularization

Regularization is a class of techniques that comes from the field of statistics (MacKay 1992a, b). In its simplest form, it consists of adding a *regularization term* to the cost function to be optimized:

$$E_{\text{total}} = E + \lambda E_{\text{reg}}$$

where E is the cost function that we defined in the previous sections, E_{reg} is the regularization term, λ is a parameter controlling the amount of regularization and E_{total} is the total cost function that will be minimized. The regularization term is chosen so that it tends to smooth the function that is generated by the network at its outputs. This term should have small values for weight vectors that generate smooth outputs, and large values for weight vectors that generate unsmooth outputs. An intuitive justification for the use of such a term can be given by considering a simple example (figure C1.2.11). Assume that a number of training data points are given (in the figure these are represented by dark circles). There is an infinite number of functions that pass through these points, two of which are represented in the figure. Of these, clearly the most reasonable are the ones that are smoothest. If the function to be approximated is smooth, then the approximator's output should be smooth also. On the other hand, if the function to be approximated is unsmooth, then only by chance would an unsmooth function generated by a network approximate the desired one, in the regions between the given data points, since unsmooth functions have a very large variability. Therefore, only by chance would the network generalize well, in such a case. Only a larger number of training points would allow us to expect to be able to successfully approximate such a function. Therefore, one should bias the training algorithm towards producing smooth output functions. This can be done through the use of a regularization term (in the theory of statistics, supervised learning can be viewed as a form of maximum-likelihood estimation, and in this context the use of a regularization term can be justified in a more elaborate way, by taking into consideration a prior distribution of weight vectors (MacKay 1992a, b)).

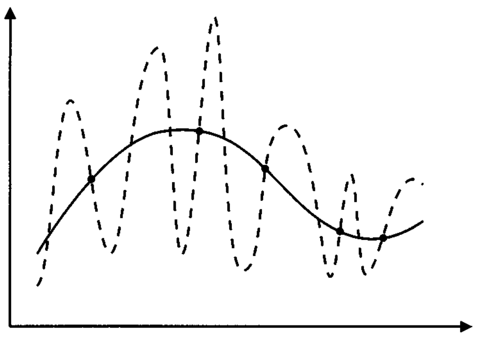

Figure C1.2.11. An illustration of generalization. Given the data points denoted by full circles, there is an infinite number of functions that pass through them. Only the smooth ones can be expected to generalize well.

One of the simplest regularization terms, which is often used in practice (Krogh and Hertz 1992), is the squared norm of the weight vector

$$E_{\text{reg}} = \sum_{j,i} w_{ji}^2 \, .$$

Use of such a regularization term is justified since smaller weights tend to produce slower-changing (and therefore smoother) functions. The use of this term leads to gradient components that are given by

$$\frac{\partial E_{\text{total}}}{\partial w_{ji}} = \frac{\partial E}{\partial w_{ji}} + \lambda w_{ji} \, .$$

The first term on the right-hand side of this equation is still computed by the backpropagation rule. Since the derivative of E_{total} is to be subtracted (after multiplication by the step size parameter) from the weight itself, we see that if the derivative of E is zero, the weight will decay exponentially to zero. For this reason, this technique is often called *exponential decay*. Other forms of regularization terms have been proposed in the literature, which are based e.g. on minimizing derivatives of the function generated by the network (Bishop 1990), or on placing a smooth cost on the individual weights, in an attempt to reduce their number (Weigend *et al* 1991).

A type of regularization term that appears to be particularly promising has been recently introduced (Williams 1994). Instead of the sum of the squares of the weights, it uses the sum of their absolute values:

$$E_{\text{reg}} = \sum_{j,i} |w_{ji}| \, .$$

Use of this term leads to

$$\frac{\partial E_{\text{total}}}{\partial w_{ji}} = \frac{\partial E}{\partial w_{ji}} + \lambda \operatorname{sgn}(w_{ji})$$

where 'sgn' denotes the sign function. If the derivative of E is zero, the weight will decay linearly to zero, reaching that value in a finite time. Only if the derivative of E relative to a weight has absolute value larger than λ will this weight be able to escape the zero value. Therefore, this E_{reg} term acts simultaneously as a regularizer, tending to keep the weights small, and as a pruner, since it automatically sets the least important weights to zero. Experience with this technique is still limited, but its ability to perform both regularization and pruning during the normal training of the network gives it a potential that should not be overlooked. We will designate this form of regularization as *linear decay*, for the reasons given above, or *Laplacian regularization*, since it can be justified, in a statistical framework, by assuming a Laplacian prior on the weights. One word of caution regarding the use of this form of regularization concerns the fact that the regularizer term E_{reg} is not differentiable relative to the weights when these have a value of zero. A way to deal with this problem is discussed in Williams (1994). A simpler way, which this author has used with success, is to check, in every training step, whether each weight has changed sign, and to set the weight to zero if it did. The weight is allowed to leave the zero value in later training steps, if $|\partial E/\partial w_{ji}| > \lambda$.

In finalizing this section, we should point out that there are several other approaches to the issue of trying to find a network with good generalization ability, and also to other related issues, such as trying to estimate the generalization ability of a given network. One of the best known of these approaches is based on the concept of *Vapnik–Chervonenkis* dimension (often designated simply *VC dimension*) (Guyon *et al* 1992). B3.5.2.2

C1.2.7 Application examples

We have already seen, in figure C1.2.9, two examples of networks trained to perform the logical XOR operation. Another artificial problem that is often used to test network training is the so-called *encoder problem*. A network with m inputs and m outputs is trained to perform an identity mapping (i.e. to yield output patterns that are equal to the respective input patterns) in a universe consisting of m patterns: those obtained by setting one of the components to 1 and all other ones to 0. The difficulty lies in the fact that the network topology that is adopted has a hidden layer with fewer than m units, forming a bottleneck. The network has to learn to encode the m patterns into different combinations of values of the hidden units, and to decode these combinations to yield the correct outputs. An example of a 4–2–4 encoder is shown in figure C1.2.12. Table C1.2.1 shows the encoding learned by a network with the topology of figure C1.2.12, trained by backpropagation. In this case target values were 0.05 and 0.95 instead of 0 and 1, respectively, as explained in section C1.2.5.2. It should be noted that, with the given architecture, the network cannot reproduce the target values exactly. This is why it sometimes outputs 0.02 and sometimes 0.06, instead of 0.05.

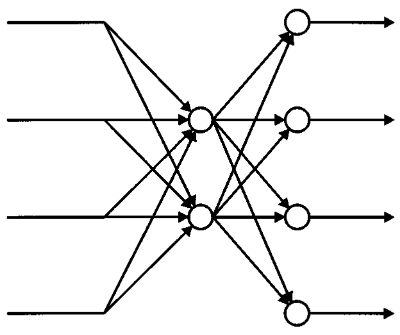

Figure C1.2.12. A 4–2–4 encoder.

Multilayer perceptrons have a rather widespread use, in very diverse application areas. We cannot give a full description of any of these applications here. We shall only give brief accounts of some of them, with references to publications where the reader can find more details.

Often, perceptrons are used as classifiers. A well-known example is the application to the *recognition of handwritten digits* (Le Cun *et al* 1990a). Normally, digit images are segmented, normalized in size and de-skewed. After this, their resolution is lowered to a manageable level (e.g. 16 × 16 pixels), before they are fed to a recognizer MLP. Recognition error rates of only a few percent can be achieved. A significant G1.3

Table C1.2.1. Encoding learned by the network of figure C1.2.12.

inputs				hidden units		outputs			
1.0	0.0	0.0	0.0	0.95	0.94	0.95	0.06	0.02	0.06
0.0	1.0	0.0	0.0	0.07	0.95	0.06	0.95	0.06	0.02
0.0	0.0	1.0	0.0	0.10	0.03	0.02	0.06	0.95	0.06
0.0	0.0	0.0	1.0	0.95	0.08	0.06	0.02	0.06	0.95

percentage of errors normally comes from the segmentation, which is not performed by neural means. In the author's group (unpublished work), an error rate of 3.8% on zipcode digits was achieved, with automatic segmentation followed by manual elimination of the few gross segmentation errors (segments with no digit at all, or with two or more complete digits). For digits that are pre-segmented, e.g. by being written in forms with boxes for individual digits, it is now possible to achieve recognition errors below 1%, a performance that is already suitable for replacing manual data entry. Several such systems are probably in use these days. The author knows of one designed and being used in Spain (López 1994). However, the problems of automatic digit segmentation and, more generally, of segmentation of cursive handwriting are still hard to deal with (Matan *et al* 1992).

Another important example of a classification application is *speech recognition*. Here, perceptrons can be used *per se* (Waibel 1989) or in hybrid systems, combined with hidden Markov models. See Robinson *et al* (1993) for an example of a state-of-the-art hybrid recognizer for large vocabulary, speaker independent, continuous speech. In hybrid systems, MLPs are actually used as probability estimators, based on an important property of supervised systems: when they are trained for classification tasks, using as cost function the quadratic error (or certain other cost functions), they essentially become estimators of the probabilities of the classes given the input vectors. This property is discussed in Richard and Lippmann (1991). In another example of a classification application, MLPs have been used to validate sensor readings in an industrial plant (Ramos *et al* 1994). F1.7.2, G1.4

In nonclassification, analog tasks, an important class is formed by *control applications*. An interesting example is that of a neural network system that is used to drive a van, controlling the steering based on an image of the road supplied by a forward-looking video camera (Pomerleau 1991). This kind of system has already been used to drive the vehicle on a highway at speeds up to 30 mph. It can also be used, with appropriately trained networks, to drive the vehicle on various other kinds of roads, including some that are hard to deal with by classical means (e.g. dirt roads covered with tree shadows) (Pomerleau 1993). F1.9

Another example of a control application is the control of fast movements of a robot arm, a problem that is hard to handle by more formal, theoretical means (Goldberg and Pearlmutter 1989). For further examples of applications to control, see White and Sage (1992). There have already been in the market, for a few years, industrial control modules that incorporate multilayer perceptrons.

Another important area of application is prediction. Multilayer perceptrons (and also other kinds of networks, namely those based on radial basis functions) have been used in the academic problem of predicting chaotic time series (Lapedes and Farber 1987), but also to predict consumptions of commodities (Yuan and Fine 1993), crucial variables in *industrial plants* (Cruz *et al* 1993) and so on. A very appealing, G2.8
but also somewhat controversial area is prediction of *financial time series* (Trippi and Turban 1993). G6.3

The practical applications of neural networks are constantly increasing in number. Given the impossibility of making an exhaustive listing here, we shall content ourselves with the above examples.

C1.2.8 Recurrent networks

Recurrent networks are networks with unit interconnections that form loops. They can be employed in two very different modes. One is nonsequential, that is, it involves no memory, the desired output for each input pattern depending only on that pattern and not on past ones. The other mode is sequential, that is, desired outputs depend not only on the current input pattern, but also on previous ones. We shall deal with them separately.

C1.2.8.1 Nonsequential networks

In this mode, as said above, desired outputs depend only on the current input pattern. Furthermore, it is assumed that whenever a pattern is presented at the network's input, it is kept fixed long enough to allow the network to reach equilibrium. As is well known from the theory of nonlinear dynamic systems (Thompson and Stewart 1986), a network with a fixed input pattern can exhibit three different kinds of behavior: it can converge to a fixed point, it can oscillate (either periodically or quasi-periodically) and it can have chaotic behavior. In what follows, we shall assume that for each input pattern the network will have stable behavior, with a single fixed point. The conditions under which this will happen are discussed later in this section.

Recurrent backpropagation. In this nonsequential situation, the gradient of the cost function E can still be computed by backward propagation of derivatives through a backpropagation network, in a natural extension of the backpropagation rule of feedforward networks (this extension is usually designated *recurrent backpropagation*). The proof of this fact was first given by Almeida (1987), and soon thereafter independently by Pineda (1987). Here we shall give a version of the proof based on graphs, which is more intuitive than the ones given in those references.

Consider first a recurrent nonlinear network N (not necessarily a multilayer perceptron), which has a single output, any number of inputs, and an internal branch which is linear with a gain w. Such a network, with the notation that we will adopt for its variables, is depicted in figure C1.2.13(a). A single input is shown, for simplicity, but multiple inputs would be treated in exactly the same manner, as we shall see. We assume that this network, as well as all other networks used in this proof, are in equilibrium at fixed points. We wish to compute the derivative of the network's output relative to w, and therefore we shall give an infinitesimal increment $\mathrm{d}w$ to w. This can be done by changing w to $w + \mathrm{d}w$, but it can also be achieved by adding an extra branch with gain $\mathrm{d}w$, as shown in figure C1.2.13(b). Of course, all internal variables, as well as the output, will suffer increments, as indicated in the figure.

The state of the network will not change if we replace the new branch by an input branch, as long as its contribution to its sink node is unchanged. This could be achieved by keeping the gain $\mathrm{d}w$ and the input $y + \mathrm{d}y$ of this branch unchanged. We can, however, change the input to y, since the contribution $\mathrm{d}y\ \mathrm{d}w$ is a higher order infinitesimum, and can therefore be disregarded (figure C1.2.13(c)).

We shall now linearize the network around its fixed point, obtaining a linear network NL that takes into account only increments (figure C1.2.13(d)). Note that the original input branch disappears, since its contribution has suffered no increment. If we had multiple inputs, the same would have happened to all of them.

We will now divide the contribution of the input branch by $\mathrm{d}w$, by changing its gain to unity. Since this network is linear, its node variables and its output will change to derivatives relative to w, which we will represent by means of upper dots, for compactness (i.e. for example, $\dot{o} = \partial o/\partial w$; see figure C1.2.13($e$)).

Finally, we will transpose the network, obtaining network NLT, shown in figure C1.2.13(f) (recall that transposition of a linear network consists in changing the direction of flow of all branches, keeping their gains; inputs become outputs, and vice-versa; summation points become divergence points, and vice-versa). From the *transposition theorem* (Oppenheim and Schafer 1975) we know that the input–output relationship of the network is not changed by transposition, i.e. if we place y at its input we will still obtain $\dot{o}$ at its output. Therefore, we can write

$$\dot{o} = ty$$

where t is the total gain from the input to the output node of the NLT network.

Now consider a recurrent perceptron P (figure C1.2.14(a)) with several outputs, and assume that we wish to compute the derivative of an output o_p relative to a weight w_{ji}. By the same reasoning, we can write

$$\dot{o}_p = t_{ip} y_j$$

where we now use the upper dot to designate the derivative relative to w_{ji}. The factor t_{ip} is the total gain of the linearized and transposed network, PLT, from input p to node i (cf figure C1.2.14(b)). Finally, let us consider the derivative of a cost function term E^k (corresponding to a given input pattern $\boldsymbol{x}^k$) relative to w_{ji}. Using the chain rule, we can write

$$\frac{\partial E^k}{\partial w_{ji}} = \sum_{p \in P} \frac{\partial E^k}{\partial o_p} \dot{o}_p$$

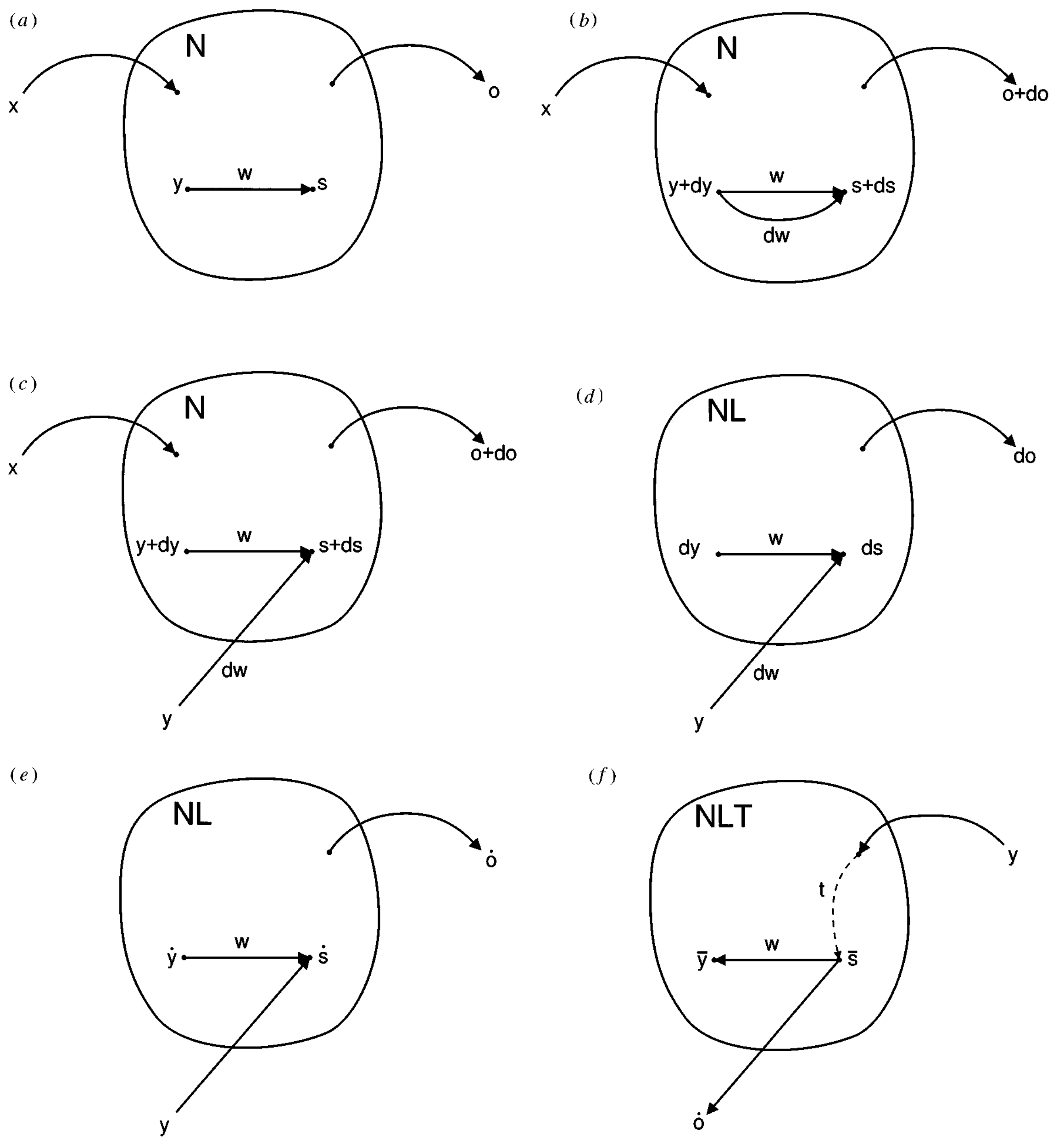

Figure C1.2.13. Illustration of the proof of validity of the backpropagation rule for recurrent networks. Case of a general network. See text for explanation.

and therefore

$$\frac{\partial E^k}{\partial w_{ji}} = \sum_{p \in P} \frac{\partial E^k}{\partial o_p} t_{ip} y_j$$
$$= y_j \sum_{p \in P} \frac{\partial E^k}{\partial o_p} t_{ip} t_{ip}$$

where P is the set of indices of units that produce outputs. Noting that network PLT is linear, we can write

$$\frac{\partial E^k}{\partial w_{ji}} = y_j \bar{s}_i \tag{C1.2.13}$$

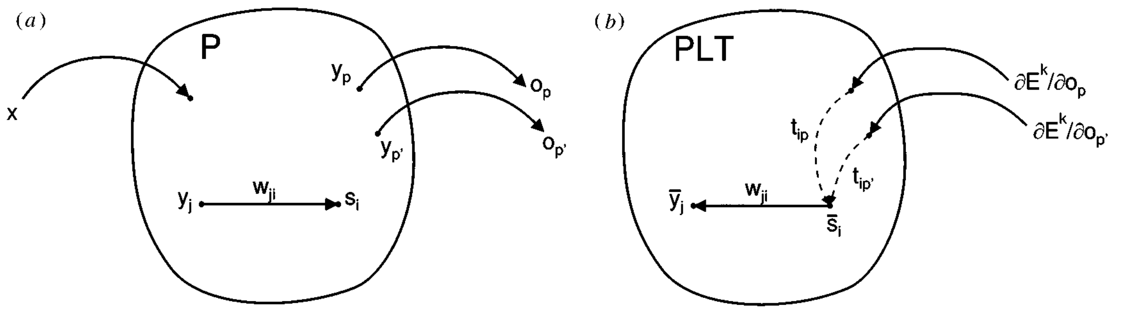

Figure C1.2.14. Illustration of the proof of validity of the backpropagation rule for recurrent networks. Case of a recurrent perceptron. See text for explanation.

where, as depicted in figure C1.2.14(*b*), $\bar{s}_i$ is obtained in the corresponding node of network PLT when the values $\partial E^k/\partial o_p$ are applied at its inputs.

If we assume that the original perceptron was feedforward, we recognize network PLT as the backpropagation network. Equation (C1.2.13) is the same as (C1.2.9), proving the validity of the backpropagation rule for feedforward networks, described in section C1.2.3.1. We will keep the designation of *backpropagation network* for network PLT in the case of recurrent networks. As we saw, this network is still obtained from the original perceptron by linearization followed by transposition. The recurrent backpropagation rule states that, if we apply the values $\partial E^k/\partial o_p$ to the corresponding inputs of the backpropagation network, the partial derivative of the cost function relative to a weight will be given by the product of the inputs of that weight's branches in the perceptron network and in the backpropagation network. Of course, the special case of the quadratic error, described in section C1.2.3.1, where one places the errors at the inputs of the backpropagation network, and then uses (C1.2.10), is also still valid in the recurrent case. For this reason, the backpropagation network is still often called the *error propagation network*, in the recurrent case.

Training a recurrent network by backpropagation takes essentially the same steps as for a feedforward network. The difference is that, when a pattern is applied to the perceptron network, this network must be allowed to stabilize before its outputs and node values are observed. The error propagation network must also be allowed to stabilize, when the derivatives $\partial E^k/\partial o_p$ are applied to its inputs. In digital implementations (including computer simulations) this involves an iteration in the propagation through the perceptron, until a stable state is found, and a similar loop in the propagation through the backpropagation network. In analog implementations the networks will evolve, through their own dynamics, to their stable states.

An important practical remark is that, in recurrent networks, the gradient's components can easily have a much larger dynamic range than in feedforward networks. The use of a technique such as adaptive step sizes, and of the robustness measures described in section C1.2.4.3, is therefore even more important here than for feedforward networks. Note that the gradient can even become infinite, at some points in weight space. This, however, does not cause any significant practical problem: gradient components can simply be limited to some convenient large value, with the proper sign.

Network stability. We assumed above that, with any fixed pattern at its input, the perceptron network was stable and had a single fixed point. It is this author's experience that often, when training recurrent networks with recurrent backpropagation, the networks that are obtained during the training process are all stable and all have single fixed points. There are exceptions, however, and it would be desirable to be able to guarantee that networks will in fact always be stable, and will always have a single fixed point. The issue of stability can be dealt with by means of a sufficient condition for stability, which we shall discuss next. The discussion of the number of fixed points will be deferred to the end of this section.

To derive a sufficient condition for stability, we first note that, while the static equations (C1.2.4) and (C1.2.5) suffice to describe the static behavior of a network, and therefore to find its fixed points, the dynamic behavior of the network is only defined if we specify the dynamic behavior of its units. Therefore, a discussion of network stability will always involve the units' dynamic behavior.

If some restrictions are imposed on it, a recurrent perceptron is formally equivalent to a *Hopfield network* with graded units (Hopfield 1984). These restrictions are that the units' dynamic behavior is as schematized in figure C1.2.15(*a*), that weights between units are symmetrical, i.e. $w_{ji} = w_{ij}$ for C1.3.4

$i, j = m+1, \ldots, N$, and that the units' nonlinearities are all increasing, bounded functions. The stability of such networks has been proved in Hopfield (1984) (we have assumed that the network variables are voltages; if currents were considered instead, then the resistor and capacitor should both be connected from the input to ground, as in Hopfield (1984)).

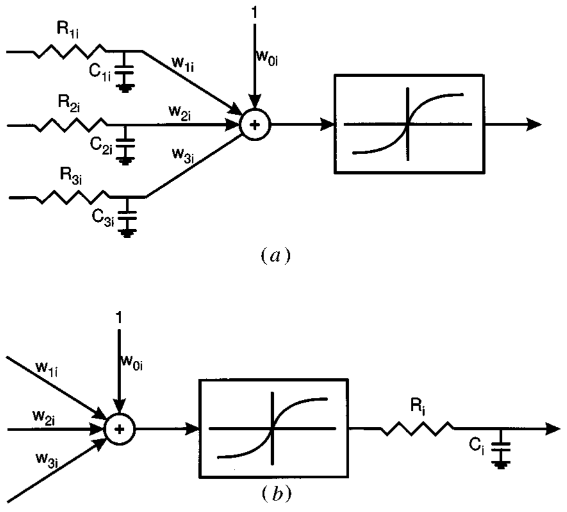

Figure C1.2.15. Typical dynamic behaviors assumed for units of continuous-time recurrent networks.

The behavior of figure C1.2.15(a) normally arises from attempting to model the dynamic behavior of biological neurons. When considering network realizations based on analog electronic systems, it is more natural to consider the dynamic behavior of figure C1.2.15(b). This is because, unless special measures are taken, an analog electronic circuit will have a lowpass behavior that can be modeled, to a first approximation, by a first-order lowpass system. The two behaviors are equivalent if all RC time constants are equal, but otherwise they are not. Here we shall give the proof of stability for the behavior of figure C1.2.15(b). This proof was first given in Almeida (1987), and is very similar to the proof given in Hopfield (1984) for the dynamic behavior of figure C1.2.15(a).

Using the notation given in figure C1.2.15(b), we can write

$$\begin{aligned} s_i &= \sum_{j=0}^{N} w_{ji} y_j \\ u_i &= S(s_i) \\ \frac{\mathrm{d}y_i}{\mathrm{d}t} &= \frac{1}{\tau_i}(u_i - y_i) \end{aligned} \tag{C1.2.14}$$

where $\tau_i = R_i C_i$ is the time constant of the RC circuit of the ith unit. Here we assume that the index i varies from $m+1$ to N, as in (C1.2.4) and (C1.2.5). We shall prove the network's stability by showing that it has a Lyapunov function (Willems 1970) that always decreases with time. The Lyapunov function that we will consider is

$$W = -\tfrac{1}{2} \sum_{j,i}^{N} w_{ji} y_i y_j + \sum_{i=m+1}^{N} U(y_i)$$

where U is a primitive of S^{-1}, the inverse of S (see figure C1.2.16). We are still assuming, as in section C1.2.3, that y_0 has a fixed value of 1, and that $y_1, \ldots, y_m$ represent the input components. We are also still assuming that the nonlinearities of all units are equal (it would again be straightforward to extend this proof to the situation in which the nonlinearities differ from one unit to another, but are all increasing and bounded; the proof could still be easily extended to the case in which all nonlinearities are decreasing and bounded; in this case the function W would increase with time, instead of decreasing).

Since we assumed that the inputs do not change, the time derivative of W is given by

$$\frac{\mathrm{d}W}{\mathrm{d}t} = \sum_{i=m+1}^{N} \frac{\mathrm{d}W}{\mathrm{d}y_i} \frac{\mathrm{d}y_i}{\mathrm{d}t} . \tag{C1.2.15}$$

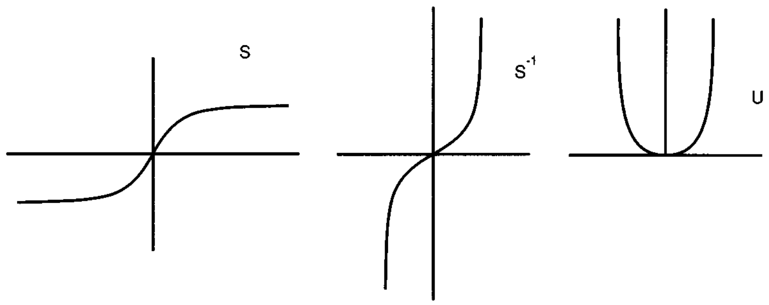

Figure C1.2.16. The functions S, S^{-1} and U. See text for explanation.

For $i = m+1, \ldots, N$, we have

$$\begin{aligned}\frac{\partial W}{\partial y_i} &= -\sum_{j=0}^{N} w_{ji} y_j + U'(y_i) \\ &= -[s_i - S^{-1}(y_i)] \\ &= -[S^{-1}(u_i) - S^{-1}(y_i)]\,.\end{aligned}$$

Since S is an increasing function, S^{-1} also is, and therefore either the difference in the last equation has the same sign as the difference in (C1.2.14), or they are simultaneously zero. Therefore, the products in (C1.2.15) are all negative or zero, and $\mathrm{d}W/\mathrm{d}t$ must be negative or zero. It is zero if and only if all the $\partial W/\partial y_i$ and the $\mathrm{d}y_i/\mathrm{d}t$ are simultaneously zero. In that case the network is in a fixed point, and W is at a point of stationarity. Since W always decreases in time during the network's evolution, the network's state cannot oscillate or have chaotic behavior. It can only move towards a fixed point, or to infinity. But since the y_i are bounded (because S is bounded), movement towards infinity is not possible, and the state must converge towards some fixed point. As we saw, these fixed points occur at the points of stationarity of W.

A useful remark (Almeida 1987) is that, except for marginally stable states, whenever the perceptron network is stable, the backpropagation network will also be stable, if the same RC-type dynamics are used in it. In fact, if the perceptron is in a nonmarginal stable state, the linearized perceptron network will also be stable. If we write its equations in the standard state space form (Willems 1970)

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \mathbf{A}\boldsymbol{u}$$

where $\boldsymbol{u}$ is the vector of state variables and $\mathbf{A}$ is the system matrix, then it will be stable if and only if all the eigenvalues of $\mathbf{A}$ have negative real parts. The backpropagation network, being the transpose of this system, has state equations

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \mathbf{A}^{\mathrm{T}}\boldsymbol{u}$$

where $\bar{\boldsymbol{u}}$ is the state vector of the backpropagation network and $\mathbf{A}^{\mathrm{T}}$ is the transpose of $\mathbf{A}$. But the eigenvalues of a matrix and of its transpose are equal. Therefore, if the linearized perceptron was stable, the backpropagation network will also be stable. Here, *transpose* is taken in the dynamic system sense. In practice this means that the RC dynamics have to be kept in the backpropagation network too.

The above remark is always true, except for marginally stable states, which are those stable states for which the linearized network is not stable. They lie at the boundary between stability and instability, and can normally be disregarded in practice, since the probability of their occurrence is essentially zero. To train a network with the guarantee that it will always be stable, we therefore have to obey three conditions.

(i) To use nonlinearities which are increasing and bounded. Networks with sigmoidal units always satisfy this condition.
(ii) To keep the weights symmetrical. For this purpose, we have first to initialize them in a symmetrical way, and then to keep them symmetrical during training. This is an example of a situation of shared weights, and is dealt with in the manner we described in section C1.2.5.5: the two derivatives

$\partial E^k/\partial w_{ij}$ and $\partial E^k/\partial w_{ji}$ are both computed using recurrent backpropagation, and their sum is used for updating both w_{ji} and w_{ij}.

(iii) To implement the RC dynamics both in the perceptron and in the backpropagation network. In digital implementations this means performing a numerical simulation of the continuous-time dynamics. If stability is not achieved, the numerical simulation is too coarse, and its time resolution should be increased. In analog implementations, RC circuits can actually be placed both in the perceptron and in the backpropagation network, to ensure that they have the appropriate dynamics.

Clearly, weight symmetry is a sufficient, but not necessary condition for stability. For example, feedforward networks are always stable, but do not obey the symmetry condition. Weight symmetry is a restriction on the network's adaptability, and it can be argued that it will reduce the network's capabilities. This is a price to be paid for being sure to obtain a network that will always be stable. But as we said at the beginning of this section, training without enforcing symmetry often yields stable networks, and in many situations it may be worth trying first, before resorting to symmetrical networks.

We come now to the discussion of the requirement that there be a single fixed point for each input pattern. Unfortunately, we do not know of any sufficient condition for guaranteeing that this will be true. The discussion of this issue can therefore only be made in qualitative terms. In practice, we have observed situations with multiple stable states only very seldom, and we never needed to take any special measures to cope with them—multiple stable states normally merged by themselves, during training. This can be explained by noting that, when training a recurrent network, we are in fact trying to move its stable states to given areas that are determined by the desired values of the outputs. If two different stable states exist for the same input pattern, and if the network stabilizes sometimes in one and sometimes in the other, then we will be trying to move them both to the same region. It is therefore not too surprising that they will merge. On the other hand, if there are multiple stable states but the network always stabilizes in the same one, then the other ones can be disregarded, as if they did not exist, since they do not influence the network's behavior in any way.

C1.2.8.2 Sequential networks

Besides the nonsequential mode described in section C1.2.8.1, recurrent networks can also be used in a sequential, or dynamic mode. In this case, network outputs depend not only on the current input, but also on previous inputs. There are several variants of the sequential mode, and we will concentrate here on the one that is most commonly used: discrete-time recurrent networks.

In this mode, it is assumed that the network's inputs only change at discrete times $t = 1, 2, \ldots,$ and that there are units in the network whose outputs are also only updated at these discrete times, synchronously with the inputs. We shall designate these units *discrete-time units*. The other units, whose outputs immediately follow any variations of their inputs, will be called *instantaneous units*. Wherever interconnections between units form loops, there must be at least one discrete-time unit in the loop. There may, however, be more than one of these units per loop. Often, people build networks in which all units are discrete-time ones, as in figure C1.2.17(a). However, nothing prevents us from using discrete-time and instantaneous units in the same network, as long as there is at least one discrete-time unit per loop. A simple example of a network with one instantaneous and two discrete-time units is given in figure C1.2.17(b). We will use this second network as an example, to better specify the operation of networks of this kind. To be consistent with the conventions used above, we will identify unit 1 with the input, that is, $y_1^n = x^n$. The input has some initial value x^0 (here, we will denote by an upper index the time step that variables refer to). Units 2 and 3, which are the discrete-time ones, have initial states y_2^0 and y_3^0 . Unit 4, which is instantaneous, immediately reflects at its output whatever is present at its input. Therefore, its output is always given by

$$y_4^n = S(w_{24}y_2^n)$$

(here n denotes the discrete time, and not the iteration number as in previous sections). Whenever a new discrete-time step arises, the input changes from x^n to x^{n+1}, and the outputs of the discrete-time units change to new values that are computed using the values of variables before that time step:

$$\begin{aligned} y_2^{n+1} &= S(w_{12}x^n + w_{32}^n y_3^n) \\ y_3^{n+1} &= S(w_{33}y_3^n + w_{43}y_4^n)\,. \end{aligned}$$

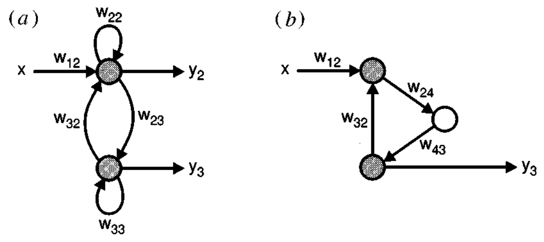

Figure C1.2.17. Examples of sequential networks. Shaded units are discrete time ones, unshaded units are instantaneous ones. (*a*) A network that has only discrete time units. (*b*) A network with both discrete time and instantaneous units.

The output of unit 4 instantaneously changes to reflect the changes of the other units and of the input:

$$y_4^{n+1} = S(w_{24} y_2^{n+1}) \, .$$

We see that, given the initial state of the network, for each input sequence $x^0, x^1, x^2, \ldots, x^{\mathrm{T}}$ the network's outputs will yield a sequence of values. The network's operation is sequential because each output value will depend on previous values of the input.

It is now easy to see why it is required that in every loop of interconnections there be at least one discrete-time unit. In a loop formed only by instantaneous units, there would be a never-ending sequence of updates, always going around the loop.

Training of this kind of recurrent network consists in finding weights so that, for given input sequences, the network approximates, as closely as possible, desired output sequences. The desired output sequences may specify target values for all time steps, or only for some of them. For example, in some situations only the desired final value of the outputs is specified. Different input sequences may be of different lengths, in which case the corresponding output sequences will also have different lengths. Naturally, training, test and validation sets will be formed by pairs of input and desired output sequences.

A great advantage of discrete-time recurrent networks is that, as we shall see, they can be reduced to feedforward networks, and can therefore be trained with ordinary backpropagation. This had already been noted in the well known book by Minsky and Papert (1969). To see how it can be done, consider again the network of figure C1.2.17(*a*). Assume that we construct a new network (figure C1.2.18(*a*)) where each unit of the recurrent network is unfolded into a sequence of units, one for each time step. Clearly, this network will always be feedforward since, in the original network, information could only flow forward in time. The input pattern of this unfolded network will be formed by the sequence of input values $x^0, x^1, x^2, \ldots, x^{\mathrm{T}}$, presented all at once to the respective input nodes. The output sequence can also be obtained all at once, from the respective output nodes. The outputs can be compared with target values (for those times for which target values do exist), and errors (or, more generally, cost function derivatives) can be fed into a backpropagation network, obtained from the feedforward network in the usual way. The only remark that needs to be made, regarding the training procedure, concerns the fact that each weight from the recurrent network appears unfolded, in the feedforward network (and also in the backpropagation network) T times. All instances of the same weight must be kept equal, since they actually correspond to a single weight in the recurrent network. This is again a situation of shared weights, that we have already seen how to handle: the derivatives relative to each of the instances of the same weight are all added together, and the sum is used to update the weight (in all its instances). Networks involving both discrete-time and instantaneous units can also be easily handled. Figure C1.2.18(*b*) shows the unfolding of the network of figure C1.2.17(*b*).

The training method that we have described is normally called *unfolding in time*, or *backpropagation through time*. It requires an amount of storage that is proportional to the number of units and to the length of the sequence being trained, since the outputs of the units at intermediate time steps must be stored until the backward propagation is completed and the cross-products of (C1.2.9) are computed. The total amount of computation per presentation of an input sequence is $\mathrm{O}(WT)$, where W is the number of weights in the network, and T is, as above, the length of the input sequence.

Unfolding in time can clearly be used in the batch and real-time modes, if real-time is understood to mean that weights are updated once per presentation of an input sequence. In some situations, instead of having a number of input sequences with the corresponding desired output sequences, one has a single very long (or even indefinitely long) input sequence, with the corresponding desired output sequence. It

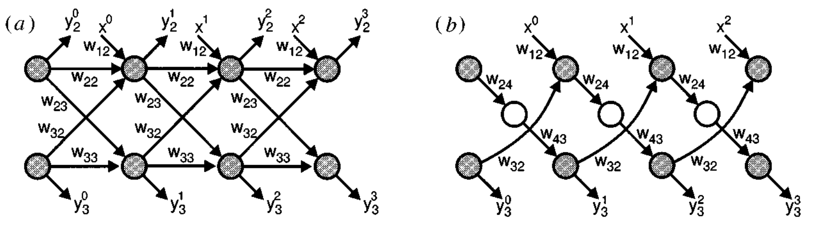

Figure C1.2.18. The unfolded networks corresponding to the sequential networks of figure C1.2.17.

would then be desirable to be able to make a weight update per time step, without having to wait for the end of the sequence to update weights. In such cases, unfolding-in-time may become rather inefficient (or even unusable, if the sequence is indefinitely long). Even in cases where there are several sequences in the training set, it might be more efficient to perform one update per time step. On the other hand, if training sequences are long, it may also be desirable not to have to store the values corresponding to all time steps, as required by the unfolding in time procedure, since these values may consume a large amount of memory. A few algorithms exist which do not need to wait for the end of the sequence to compute contributions to gradients, and which require only a limited amount of memory, irrespective of the length of the input sequence. We will mention only the best known one, often designated *real-time recurrent learning* (RTRL), which was originally proposed by Robinson and Fallside (1987) under the name of *infinite impulse response algorithm*, and is best known from later publications of Williams and Zipser (1989). This algorithm carries forward, in time, the information that is necessary to compute the derivatives of the cost function, and therefore does not need to store previous network states, and also does not need to perform backward propagations in time. There are two prices to be paid for this. One is computational complexity. While, for a fully interconnected network with N units (and therefore $W = N^2$ weights) unfolding in time requires $\mathrm{O}(N^2T)$ operations per sequence presentation, RTRL requires $\mathrm{O}(N^4T)$ operations. This quickly makes it impractical for large networks. The other price to be paid is that, if weight updates are performed at every time step, what is computed is only an approximation to the actual gradient of the cost function. Depending on the situation, this approximation may be good or bad. For some problems this is of little importance, but for others it may affect convergence, and even lead the training process to converge to wrong solutions. A variant of RTRL that deserves mentioning is called the *Green's function algorithm* (Sun *et al* 1992). It has the advantage of reducing the number of operations to $\mathrm{O}(N^3T)$. However, in numerical implementations it involves an approximation that may affect its validity for long sequences.

Several examples of the application of unfolding in time to the training of recurrent networks have appeared in the literature. A very interesting one is described in Nguyen and Widrow (1990), where a controller is trained to park a truck with a trailer in backward motion. A very early example of an application to speech was given in Watrous (1987). Examples of the use of RTRL have also appeared in the literature; for example, for the learning of grammars (Giles *et al* 1992).

Besides the discrete-time mode, recurrent networks are also sometimes used in a continuous-time mode. In this case, the outputs of units change continuously in time according to given dynamics. Inputs and target outputs of the network are then both functions of continuous time, instead of being sequences. A training algorithm for this kind of network, which is an extension of unfolding in time to the continuous time situation, was presented in Pearlmutter (1989).

C1.2.8.3 Time-delay neural networks

An architecture that is often used for sequential applications is shown in figure C1.2.19. It consists of a feedforward neural network that is fed by a delay line which stores past values of the input. In this case the sequential capabilities of the system do not come from the neural network itself, which is a plain feedforward one. They come, instead, from the delay line. An advantage of this structure is that it can be trained with standard backpropagation, since the neural network is feedforward. The disadvantages come from the facts that the architecture is not recursive and that its memory capabilities are fixed and cannot be adapted by training. For several kinds of problems, like those involving a long-time memory, this architecture may need many more weights (and therefore many more training patterns) than a recurrent one. Systems of this kind are often designated *time-delay neural networks* (TDNN). They have been applied to several kinds of problems. See Waibel (1989) for an example of an application to *speech recognition*, in which this architecture is extended by using delay lines at multiple levels, with multiple time resolutions. F1.7.2

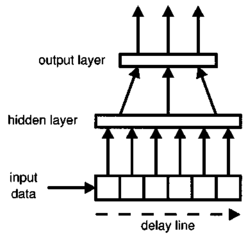

Figure C1.2.19. A time-delay neural network.

Acknowledgement

We wish to acknowledge the use of the 'United States Postal Service Office of Advanced Technology Handwritten ZIP Code Data Base (1987)', made available by the Office of Advanced Technology, United States Postal Service.

References

Almeida L B 1987 A learning rule for asynchronous perceptrons with feedback in a combinatorial environment *Proc. IEEE First Int. Conf. on Neural Networks* (New York: IEEE Press) pp 609–18

Battiti R 1992 First- and second-order methods for learning: between steepest descent and Newton's method *Neural Comput.* **4** 141–66

Becker S and Le Cun Y 1989 Improving the convergence of back-propagation learning with second order methods *Proc. 1988 Connectionist Models Summer School* ed D Touretzky, G Hinton and T Sejnowski (San Mateo, CA: Morgan Kaufmann) pp 29–37

Bishop C M 1990 Curvature-driven smoothing in backpropagation neural networks *Technical Report CLM-P-880* (Abingdon, UK: AEA Technology, Culham Laboratory)

Bryson A E and Ho Y C 1969 *Applied Optimal Control* (New York: Blaisdell)

Cruz C S, Rodriguez F, Dorronsoro J R and López V 1993 Nonlinear dynamical system modelling and its integration in intelligent control *Proc. Workshop on Integration in Real-Time Intelligent Control Systems* (Miraflores de la Sierra) pp 30-1 to 30-9

Cybenko G 1989 Approximation by superpositions of a sigmoidal function *Math. Control, Signal Syst.* **2** 303–14

Fahlman S E 1989 Fast-learning variations on back-propagation: an empirical study *Proc. 1988 Connectionist Models Summer School* ed D Touretzky, G Hinton and T Sejnowski (San Mateo, CA: Morgan Kaufmann) pp 38–51

Fahlman S E and Lebiere C 1990 The cascade-correlation learning architecture *Advances in Neural Information Processing Systems 2* ed D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 524–32

Frean M 1990 The upstart algorithm: a method for constructing and training feedforward neural networks *Neural Comput.* **2** 198–209

Funahashi K 1989 On the approximate realization of continuous mappings by neural networks *Neural Networks* **2** 183–92

Giles C L, Miller C B, Chen D, Sun G Z, Chen H H and Lee Y C 1992 Extracting and learning an unknown grammar with recurrent neural networks *Advances in Neural Information Processing Systems 4* ed J E Moody, S J Hanson and R P Lippmann (San Mateo, CA: Morgan Kaufmann) pp 317–24

Goldberg K Y and Pearlmutter B A 1989 Using backpropagation with temporal windows to learn the dynamics of the CMU direct-drive arm II *Advances in Neural Information Processing Systems 1* ed D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 356–65

Golub G H and Van Loan C F 1983 *Matrix Computations* (Baltimore, MD: Johns Hopkins University Press)

Guyon I, Vapnik V, Boser B, Bottou L and Solla S A 1992 Structural risk minimization for character recognition *Advances in Neural Information Processing Systems 4* ed J Moody, S J Hanson and Lippmann R P (San Mateo, CA: Morgan Kaufmann) pp 471–9

Hassibi B, Stork D G and Wolff G J 1993 Optimal brain surgeon and general network pruning *Proc. IEEE Int. Conf. on Neural Networks* (San Francisco, CA) pp 293–9

Hopfield J J 1984 Neurons with graded response have collective computational properties like those of two-state neurons *Proc. Natl Acad. Sci. USA 81* 3088–92

Hornik K, Sithcombe M and White H 1989 Multilayer feedforward networks are universal approximators *Neural Networks* **2** 359–66

Jacobs R 1988 Increased rates of convergence through learning rate adaptation *Neural Networks* **1** 295–307

Krogh A and Hertz J A 1992 A simple weight decay can improve generalization *Advances in Neural Information Processing Systems 4* ed J E Moody, S J Hanson and R P Lippmann (San Mateo, CA: Morgan Kaufmann) pp 950–7

Lapedes A S and Farber R 1987 Nonlinear signal processing using neural networks: prediction and system modelling *Technical Report LA-UR-87-2662* (Los Alamos, NM: Los Alamos National Laboratory)

Le Cun Y 1985 Une procédure d'apprentissage pour réseau à seuil assymétrique *Cognitiva* **85** 599–604

Le Cun Y, Boser B, Denker J S, Henderson D, Howard R E, Hubbard W and Jackel L D 1990a Handwritten digit recognition with a backpropagation network *Advances in Neural Information Processing Systems 2* ed D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 396–409

Le Cun Y, Denker J S and Solla S 1990b Optimal brain damage *Advances in Neural Information Processing Systems 2* ed D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 598–605

Le Cun Y, Kanter I and Solla S 1991 Second order properties of error surfaces: learning time and generalization *Advances in Neural Information Processing Systems 3* ed R P Lippmann, J E Moody and D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 918–24

Ljung L 1978 Strong convergence of a stochastic approximation algorithm *Ann. Statistics* **6** 680–96

López V 1994 Private communication

MacKay D J 1992a Bayesian interpolation *Neural Comput.* **4** 415–47

MacKay D J 1992b A practical bayesian framework for backprop networks *Neural Comput.* **4** 448–72

Matan O, Burges C J, Le Cun Y and Denker J S 1992 Multi-digit recognition using a space displacement neural network *Advances in Neural Information Processing Systems 4* ed J E Moody, S J Hanson and R P Lippmann (San Mateo, CA: Morgan Kaufmann) pp 488–95

Mézard M and Nadal J P 1989 Learning in feedforward layered networks: the tiling algorithm *J. Phys. A: Math. Gen.* **22** 2191–204

Minsky M L and Papert S A 1969 *Perceptrons* (Cambridge, MA: MIT Press)

Moller M F 1990 A scaled conjugated gradient algorithm for fast supervised learning *Preprint PB-339* (Aarhus, Denmark: Computer Science Department, University of Aarhus)

Mozer M C and Smolensky P 1989 Skeletonization: a technique for trimming the fat from a network via relevance assignment *Report CU-CS-421-89* (Boulder, CO: Department of Computer Science, University of Colorado)

Nguyen D and Widrow B 1990 The truck backer-upper: an example of self-learning in neural networks *Advanced Neural Computers* ed R Eckmiller (Amsterdam: Elsevier) pp 11–20

Oppenheim A V and Schafer R W 1975 *Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall)

Parker D B 1985 Learning logic *Technical Report TR-47* (Cambridge, MA: Center for Computational Research in Economics and Management Science, MIT)

Pineda F J 1987 Generalization of backpropagation to recurrent neural networks *Phys. Rev. Lett.* **59** 2229–32

Pearlmutter B A 1989 Learning state space trajectories in recurrent neural networks *Neural Comput.* **1** 263–9

Pomerleau D A 1991 Efficient training of artificial neural networks for autonomous navigation *Neural Comput.* **3** 89–97

Pomerleau D A 1993 Input reconstruction reliability estimation *Advances in Neural Information Processing Systems 5* ed S J Hanson, J D Cowan and C L Giles (San Mateo, CA: Morgan Kaufmann) pp 279–86

Press W H, Flannery B P, Teukolsky S A and Vetterling W T 1986 *Numerical Recipes* (Cambridge: Cambridge University Press)

Ramos H S, Langlois T, Xufre G, Amaral J D, Almeida L B and Silva F M 1994 Neural networks in industrial modeling and fault detection Proc. *Workshop on Artificial Intelligence in Real-Time Control (Valencia)*

Richard M D and Lippmann R P 1991 Neural network classifiers estimate Bayesian *a posteriori* probabilities *Neural Comput.* **3** 461–83

Robinson A J and Fallside F 1987 The utility driven dynamic error propagation network *Technical Report CUED/F-INFENG/TR.1* (Cambridge, UK: Cambridge University Engineering Department)

Robinson A J *et al* 1993 A neural network based, speaker independent, large vocabulary, continuous speech recognition system: the Wernicke project *Proc. Eurospeech'93 Conf. (Berlin)* pp 1941–4

Rumelhart D E, Hinton G E and Williams R J 1986 Learning internal representations by error propagation *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* vol 1 ed D E Rumelhart, J L McClelland and the PDP research group (Cambridge, MA: MIT Press) pp 318–62

Silva F M and Almeida L B 1990a Acceleration techniques for the backpropagation algorithm *Neural Networks* ed L B Almeida and C J Wellekens (Berlin: Springer) pp 110–19

Silva F M and Almeida L B 1990b Speeding up backpropagation *Advanced Neural Computers* ed R Eckmiller (Amsterdam: Elsevier) pp 151–60

Silva F M and Almeida L B 1991 Speeding-Up backpropagation by data orthonormalization *Artificial Neural Networks* vol 2, ed T Kohonen, K Mäkisara, O Simula and J Kangas (Amsterdam: Elsevier) pp 149–56

Sun G Z, Chen H H and Lee Y C 1992 Green's function method for fast on-line learning algorithm of recurrent neural networks *Advances in Neural Information Processing Systems 4* ed J E Moody, S J Hanson and R P Lippmann (San Mateo, CA: Morgan Kaufmann) pp 333–40

Thompson J M and Stewart H B 1986 *Nonlinear Dynamics and Chaos* (Chichester: Wiley)

Tollenaere T 1990 SuperSAB: fast adaptive back propagation with good scaling properties *Neural Networks* **3** 561–74

Trippi R R and Turban E (eds) 1993 *Neural Networks in Finance and Investing* (Chicago, IL: Probus)

Waibel A 1989 Modular construction of time-delay neural networks for speech recognition *Neural Comput.* **1** 39–46

Watrous R L 1987 Learning phonetic features using connectionist networks: an experiment in speech recognition *Proc. IEEE 1st International Conf. on Neural Networks* (New York: IEEE Press) pp 381–7

Weigend A S, Rumelhart D E and Huberman B A 1991 Generalization by weight-elimination with application to forecasting *Advances in Neural Information Processing Systems 3* ed R P Lippmann, J E Moody and D S Touretzky (San Mateo, CA: Morgan Kaufmann) pp 875–82

Werbos P J 1974 Beyond regression: new tools for prediction and analysis in the behavioral sciences *PhD Thesis* (Cambridge, MA: Harvard University)

White D A and Sage D A (eds) 1992 *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches* (New York: Van Nostrand Reinhold)

Widrow B and Stearns S D 1985 *Adaptive Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall)

Willems J L 1970 *Stability Theory of Dynamical Systems* (London: Thomas Nelson)

Williams P M 1994 Bayesian regularization and pruning using a Laplace prior *Cognitive Science Research Paper CSRP-312* (Brighton: School of Cognitive and Computing Sciences, University of Sussex)

Williams R J and Zipser D 1989 A learning algorithm for continually running fully recurrent neural networks *Neural Comput.* **1** 270–80

Yuan J L and Fine T L 1993 Forecasting demand for electric power *Advances in Neural Information Processing Systems 5* ed S J Hanson, J D Cowan and C L Giles (San Mateo, CA: Morgan Kaufmann) pp 739–46