7

The Backpropagation Algorithm

7.1 Learning as gradient descent

We saw in the last chapter that multilayered networks are capable of computing a wider range of Boolean functions than networks with a single layer of computing units. However the computational effort needed for finding the correct combination of weights increases substantially when more parameters and more complicated topologies are considered. In this chapter we discuss a popular learning method capable of handling such large learning problems — *the backpropagation algorithm*. This numerical method was used by different research communities in different contexts, was discovered and rediscovered, until in 1985 it found its way into connectionist AI mainly through the work of the PDP group [382]. It has been one of the most studied and used algorithms for neural networks learning ever since.

In this chapter we present a proof of the backpropagation algorithm based on a graphical approach in which the algorithm reduces to a graph labeling problem. This method is not only more general than the usual analytical derivations, which handle only the case of special network topologies, but also much easier to follow. It also shows how the algorithm can be efficiently implemented in computing systems in which only local information can be transported through the network.

7.1.1 Differentiable activation functions

The backpropagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons,

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too. One of the more popular activation functions for backpropagation networks is the *sigmoid*, a real function $s_c : \mathbb{R} \to (0, 1)$ defined by the expression

$$s_c(x) = \frac{1}{1 + \mathrm{e}^{-cx}}.$$

The constant c can be selected arbitrarily and its reciprocal $1/c$ is called the temperature parameter in stochastic neural networks. The shape of the sigmoid changes according to the value of c, as can be seen in Figure 7.1. The graph shows the shape of the sigmoid for $c = 1$, $c = 2$ and $c = 3$. Higher values of c bring the shape of the sigmoid closer to that of the step function and in the limit $c \to \infty$ the sigmoid converges to a step function at the origin. In order to simplify all expressions derived in this chapter we set $c = 1$, but after going through this material the reader should be able to generalize all the expressions for a variable c. In the following we call the sigmoid $s_1(x)$ just $s(x)$.

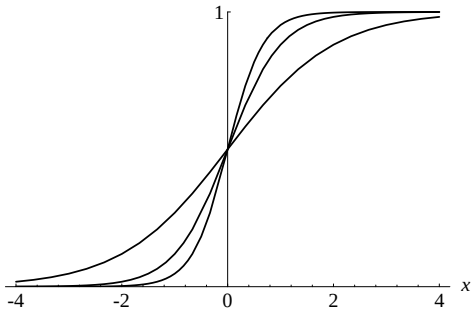

Fig. 7.1. Three sigmoids (for $c = 1$, $c = 2$ and $c = 3$)

The derivative of the sigmoid with respect to x, needed later on in this chapter, is

$$\frac{d}{dx}s(x) = \frac{\mathrm{e}^{-x}}{(1 + \mathrm{e}^{-x})^2} = s(x)(1 - s(x)).$$

We have already shown that, in the case of perceptrons, a symmetrical activation function has some advantages for learning. An alternative to the sigmoid is the symmetrical sigmoid $S(x)$ defined as

$$S(x) = 2s(x) - 1 = \frac{1 - \mathrm{e}^{-x}}{1 + \mathrm{e}^{-x}}.$$

This is nothing but the hyperbolic tangent for the argument $x/2$ whose shape is shown in Figure 7.2 (upper right). The figure shows four types of continuous "squashing" functions. The ramp function (lower right) can also be used in

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

learning algorithms taking care to avoid the two points where the derivative is undefined.

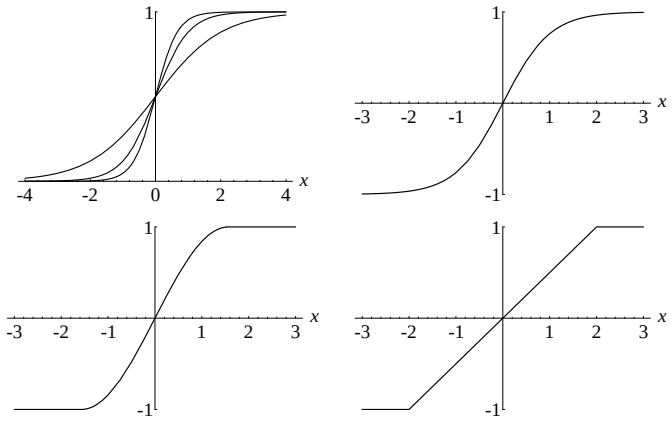

Fig. 7.2. Graphics of some "squashing" functions

Many other kinds of activation functions have been proposed and the backpropagation algorithm is applicable to all of them. A differentiable activation function makes the function computed by a neural network differentiable (assuming that the integration function at each node is just the sum of the inputs), since the network itself computes only function compositions. The error function also becomes differentiable.

Figure 7.3 shows the smoothing produced by a sigmoid in a step of the error function. Since we want to follow the gradient direction to find the minimum of this function, it is important that no regions exist in which the error function is completely flat. As the sigmoid always has a positive derivative, the slope of the error function provides a greater or lesser descent direction which can be followed. We can think of our search algorithm as a physical process in which a small sphere is allowed to roll on the surface of the error function until it reaches the bottom.

7.1.2 Regions in input space

The sigmoid's output range contains all numbers strictly between 0 and 1. Both extreme values can only be reached asymptotically. The computing units considered in this chapter evaluate the sigmoid using the net amount of excitation as its argument. Given weights $w_1, \ldots, w_n$ and a bias $-\theta$, a sigmoidal unit computes for the input $x_1, \ldots, x_n$ the output

$$\frac{1}{1 + \exp\left(\sum_{i=1}^{n} w_i x_i - \theta\right)}.$$

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

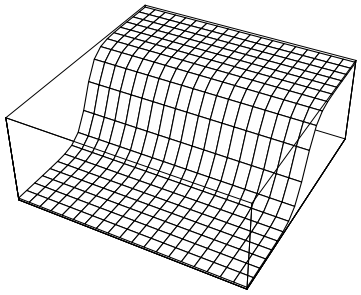

Fig. 7.3. A step of the error function

A higher net amount of excitation brings the unit's output nearer to 1. The continuum of output values can be compared to a division of the input space in a continuum of classes. A higher value of c makes the separation in input space sharper.

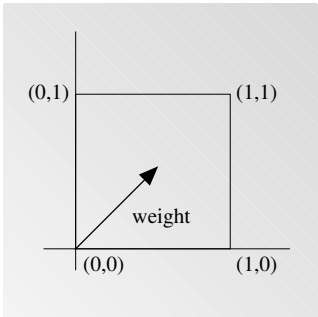

Fig. 7.4. Continuum of classes in input space

Note that the step of the sigmoid is normal to the vector $(w_1, \ldots, w_n, -\theta)$ so that the weight vector points in the direction in extended input space in which the output of the sigmoid changes faster.

7.1.3 Local minima of the error function

A price has to be paid for all the positive features of the sigmoid as activation function. The most important problem is that, under some circumstances, local minima appear in the error function which would not be there if the step function had been used. Figure 7.5 shows an example of a local minimum with a higher error level than in other regions. The function was computed for a single unit with two weights, constant threshold, and four input-output patterns in the training set. There is a valley in the error function and if

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

gradient descent is started there the algorithm will not converge to the global minimum.

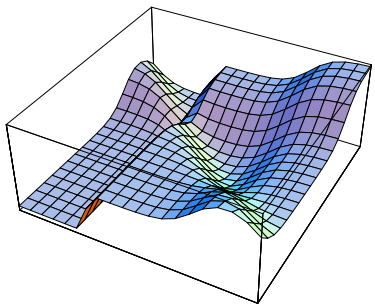

Fig. 7.5. A local minimum of the error function

In many cases local minima appear because the targets for the outputs of the computing units are values other than 0 or 1. If a network for the computation of XOR is trained to produce 0.9 at the inputs (0,1) and (1,0) then the surface of the error function develops some protuberances, where local minima can arise. In the case of binary target values some local minima are also present, as shown by Lisboa and Perantonis who analytically found all local minima of the XOR function [277].

7.2 General feed-forward networks

In this section we show that backpropagation can easily be derived by linking the calculation of the gradient to a graph labeling problem. This approach is not only elegant, but also more general than the traditional derivations found in most textbooks. General network topologies are handled right from the beginning, so that the proof of the algorithm is not reduced to the multilayered case. Thus one can have it both ways, more general yet simpler [375].

7.2.1 The learning problem

Recall that in our general definition a feed-forward neural network is a computational graph whose nodes are computing units and whose directed edges transmit numerical information from node to node. Each computing unit is capable of evaluating a single primitive function of its input. In fact the network represents a chain of function compositions which transform an input to an output vector (called a pattern). The network is a particular implementation of a composite function from input to output space, which we call the *network function*. The learning problem consists of finding the optimal combination

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

of weights so that the network function φ approximates a given function f as closely as possible. However, we are not given the function f *explicitly* but only implicitly through some examples.

Consider a feed-forward network with n input and m output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $\{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_p, \mathbf{t}_p)\}$ consisting of p ordered pairs of n- and m-dimensional vectors, which are called the input and output patterns. Let the primitive functions at each node of the network be continuous and differentiable. The weights of the edges are real numbers selected at random. When the input pattern $\mathbf{x}_i$ from the training set is presented to this network, it produces an output $\mathbf{o}_i$ different in general from the target $\mathbf{t}_i$. What we want is to make $\mathbf{o}_i$ and $\mathbf{t}_i$ identical for $i = 1, \ldots, p$, by using a learning algorithm. More precisely, we want to minimize the error function of the network, defined as

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\mathbf{o}_i - \mathbf{t}_i\|^2.$$

After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to *interpolate*. The network must recognize whether a new input vector is similar to learned patterns and produce a similar output.

The backpropagation algorithm is used to find a local minimum of the error function. The network is initialized with randomly chosen weights. The gradient of the error function is computed and used to correct the initial weights. Our task is to compute this gradient recursively.

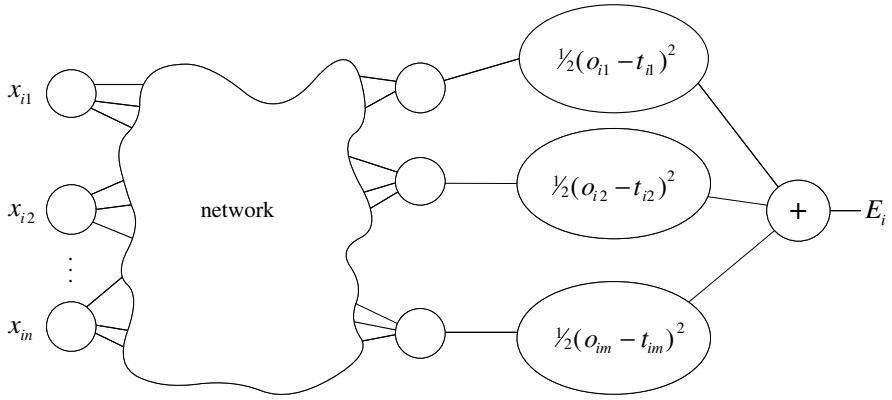

Fig. 7.6. Extended network for the computation of the error function

The first step of the minimization process consists of extending the network, so that it computes the error function automatically. Figure 7.6 shows

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

how this is done. Every one of the j output units of the network is connected to a node which evaluates the function $\frac{1}{2}(o_{ij} - t_{ij})^2$, where o_{ij} and t_{ij} denote the j-th component of the output vector $\mathbf{o}_i$ and of the target $\mathbf{t}_i$. The outputs of the additional m nodes are collected at a node which adds them up and gives the sum E_i as its output. The same network extension has to be built for each pattern $\mathbf{t}_i$. A computing unit collects all quadratic errors and outputs their sum $E_1 + \cdots + E_p$. The output of this extended network is the error function E.

We now have a network capable of calculating the total error for a given training set. The weights in the network are the only parameters that can be modified to make the quadratic error E as low as possible. Because E is calculated by the extended network exclusively through composition of the node functions, it is a continuous and differentiable function of the ℓ weights $w_1, w_2, \ldots, w_\ell$ in the network. We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\ell}).$$

Each weight is updated using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \quad \text{for } i = 1, \ldots, \ell,$$

where γ represents a learning constant, i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

With this extension of the original network the whole learning problem now reduces to the question of calculating the gradient of a network function with respect to its weights. Once we have a method to compute this gradient, we can adjust the network weights iteratively. In this way we expect to find a minimum of the error function, where $\nabla E = 0$.

7.2.2 Derivatives of network functions

Now forget everything about training sets and learning. Our objective is to find a method for efficiently calculating the gradient of a one-dimensional network function according to the weights of the network. Because the network is equivalent to a complex chain of function compositions, we expect the chain rule of differential calculus to play a major role in finding the gradient of the function. We take account of this fact by giving the nodes of the network a composite structure. Each node now consists of a left and a right side, as shown in Figure 7.7. We call this kind of representation a *B-diagram* (for backpropagation diagram).

The right side computes the primitive function associated with the node, whereas the left side computes the derivative of this primitive function for the same input.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

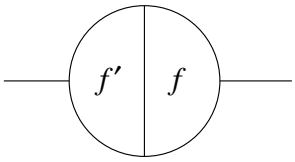

Fig. 7.7. The two sides of a computing unit

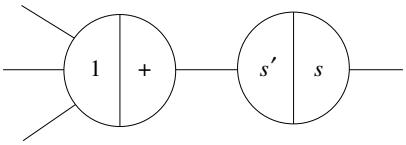

Fig. 7.8. Separation of integration and activation function

Note that the integration function can be separated from the activation function by splitting each node into two parts, as shown in Figure 7.8. The first node computes the sum of the incoming inputs, the second one the activation function s. The derivative of s is s' and the partial derivative of the sum of n arguments with respect to any one of them is just 1. This separation simplifies our discussion, as we only have to think of a single function which is being computed at each node and not of two.

The network is evaluated in two stages: in the first one, the feed-forward step, information comes from the left and each unit evaluates its primitive function f in its right side as well as the derivative f' in its left side. Both results are stored in the unit, but only the result from the right side is transmitted to the units connected to the right. The second step, the backpropagation step, consists in running the whole network backwards, whereby the stored results are now used. There are three main cases which we have to consider.

First case: function composition

The B-diagram of Figure 7.9 contains only two nodes. In the feed-forward step, incoming information into a unit is used as the argument for the evaluation of the node's primitive function and its derivative. In this step the network computes the composition of the functions f and g. Figure 7.10 shows the state of the network after the feed-forward step. The correct result of the function composition has been produced at the output unit and each unit has stored some information on its left side.

In the backpropagation step the input from the right of the network is the constant 1. Incoming information to a node is *multiplied* by the value stored in its left side. The result of the multiplication is transmitted to the next unit to the left. We call the result at each node the *traversing value* at this node. Figure 7.11 shows the final result of the backpropagation step, which is $f'(g(x))g'(x)$, i.e., the derivative of the function composition $f(g(x))$

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

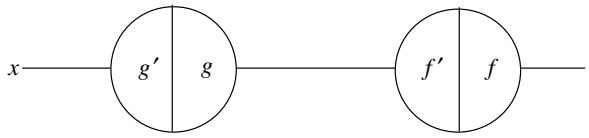

Fig. 7.9. Network for the composition of two functions

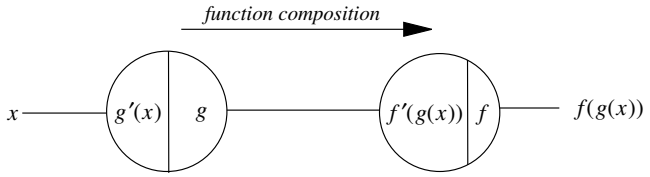

Fig. 7.10. Result of the feed-forward step

implemented by this network. The backpropagation step provides an implementation of the chain rule. Any sequence of function compositions can be evaluated in this way and its derivative can be obtained in the backpropagation step. We can think of the network as being used backwards with the input 1, whereby at each node the product with the value stored in the left side is computed.

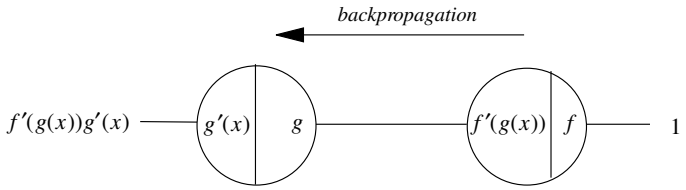

Fig. 7.11. Result of the backpropagation step

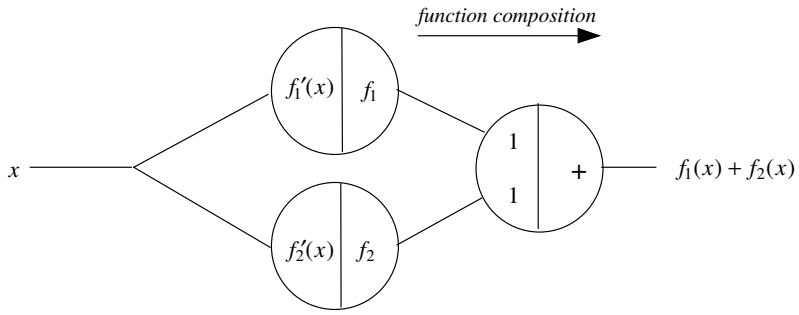

Fig. 7.12. Addition of functions

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Second case: function addition

The next case to consider is the addition of two primitive functions. Figure 7.12 shows a network for the computation of the addition of the functions f_1 and f_2 . The additional node has been included to handle the addition of the two functions. The partial derivative of the addition function with respect to any one of the two inputs is 1. In the feed-forward step the network computes the result $f_1(x) + f_2(x)$. In the backpropagation step the constant 1 is fed from the left side into the network. All incoming edges to a unit fan out the traversing value at this node and distribute it to the connected units to the left. Where two right-to-left paths meet, the computed traversing values are added. Figure 7.13 shows the result $f_1'(x) + f_2'(x)$ of the backpropagation step, which is the derivative of the function addition $f_1 + f_2$ evaluated at x. A simple proof by induction shows that the derivative of the addition of any number of functions can be handled in the same way.

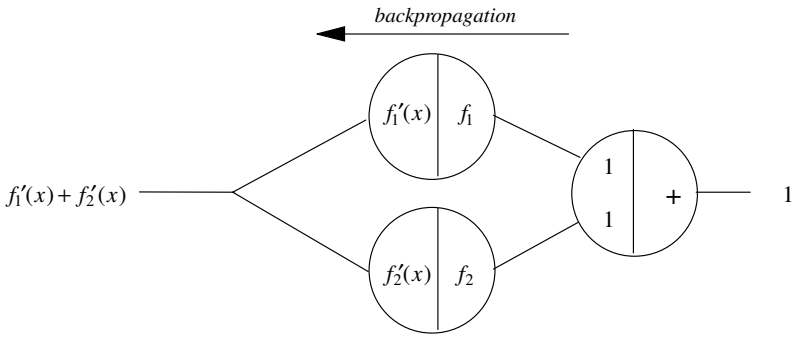

Fig. 7.13. Result of the backpropagation step

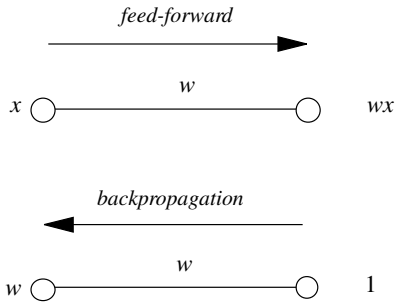

Fig. 7.14. Forward computation and backpropagation at an edge

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Third case: weighted edges

Weighted edges could be handled in the same manner as function compositions, but there is an easier way to deal with them. In the feed-forward step the incoming information x is multiplied by the edge's weight w. The result is wx. In the backpropagation step the traversing value 1 is multiplied by the weight of the edge. The result is w, which is the derivative of wx with respect to x. From this we conclude that weighted edges are used in exactly the same way in both steps: they modulate the information transmitted in each direction by multiplying it by the edges' weight.

7.2.3 Steps of the backpropagation algorithm

We can now formulate the complete backpropagation algorithm and prove by induction that it works in arbitrary feed-forward networks with differentiable activation functions at the nodes. We assume that we are dealing with a network with a single input and a single output unit.

Algorithm 7.2.1 *Backpropagation algorithm.*

Consider a network with a single real input x and network function F. The derivative $F'(x)$ is computed in two phases:

Feed-forward: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.

Backpropagation: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.

The following proposition shows that the algorithm is correct.

Proposition 11. *Algorithm 7.2.1 computes the derivative of the network function F with respect to the input x correctly.*

Proof. We have already shown that the algorithm works for units in series, units in parallel and also for weighted edges. Let us make the induction assumption that the algorithm works for any feed-forward network with n or fewer nodes. Consider now the B-diagram of Figure 7.15, which contains $n+1$ nodes. The feed-forward step is executed first and the result of the single output unit is the network function F evaluated at x. Assume that m units, whose respective outputs are $F_1(x), \ldots, F_m(x)$ are connected to the output unit. Since the primitive function of the output unit is φ, we know that

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

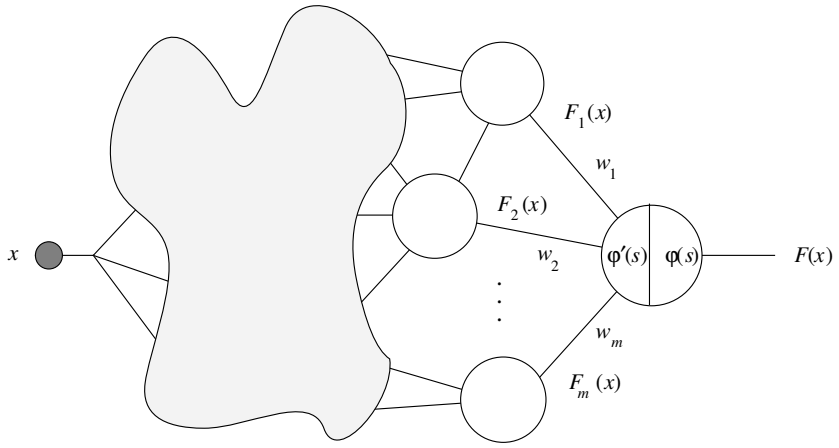

Fig. 7.15. Backpropagation at the last node

$$F(x) = \varphi(w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x)).$$

The derivative of F at x is thus

$$F'(x) = \varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x)),$$

where $s = \varphi(w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x))$. The subgraph of the main graph which includes all possible paths from the input unit to the unit whose output is $F_1(x)$ defines a subnetwork whose network function is F_1 and which consists of n or fewer units. By the induction assumption we can calculate the derivative of F_1 at x, by introducing a 1 into the unit and running the subnetwork backwards. The same can be done with the units whose outputs are $F_2(x), \ldots, F_m(x)$. If instead of 1 we introduce the constant $\varphi'(s)$ and multiply it by w_1 we get $w_1 F_1'(x)\varphi'(s)$ at the input unit in the backpropagation step. Similarly we get $w_2 F_2'(x)\varphi'(s), \ldots, w_m F_m'(x)\varphi'(s)$ for the rest of the units. In the backpropagation step with the whole network we add these m results and we finally get

$$\varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x))$$

which is the derivative of F evaluated at x. Note that introducing the constants $w_1\varphi'(s), \ldots, w_m\varphi'(s)$ into the m units connected to the output unit can be done by introducing a 1 into the output unit, multiplying by the stored value $\varphi'(s)$ and distributing the result to the m units through the edges with weights $w_1, w_2, \ldots, w_m$. We are in fact running the network backwards as the backpropagation algorithm demands. This means that the algorithm works with networks of $n + 1$ nodes and this concludes the proof. □

Implicit in the above analysis is that all inputs to a node are added before the one-dimensional activation function is computed. We can consider

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

also activation functions f of several variables, but in this case the left side of the unit stores all partial derivatives of f with respect to each variable. Figure 7.16 shows an example for a function f of two variables x_1 and x_2, delivered through two different edges. In the backpropagation step each stored partial derivative is multiplied by the traversing value at the node and transmitted to the left *through its own edge*. It is easy to see that backpropagation still works in this more general case.

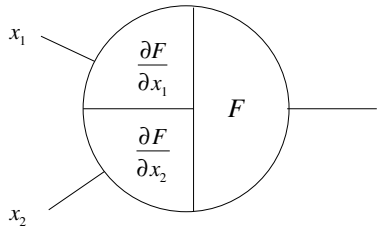

Fig. 7.16. Stored partial derivatives at a node

The backpropagation algorithm also works correctly for networks with more than one input unit in which several independent variables are involved. In a network with two inputs for example, where the independent variables x_1 and x_2 are fed into the network, the network result can be called $F(x_1, x_2)$. The network function now has two arguments and we can compute the partial derivative of F with respect to x_1 or x_2. The feed-forward step remains unchanged and all left side slots of the units are filled as usual. However, in the backpropagation step we can identify two subnetworks: one consists of all paths connecting the first input unit to the output unit and another of all paths from the second input unit to the output unit. By applying the backpropagation step in the first subnetwork we get the partial derivative of F with respect to x_1 at the first input unit. The backpropagation step on the second subnetwork yields the partial derivative of F with respect to x_2 at the second input unit. Note that we can overlap both computations and perform a single backpropagation step over the whole network. We still get the same results.

7.2.4 Learning with backpropagation

We consider again the learning problem for neural networks. Since we want to minimize the error function E, which depends on the network weights, we have to deal with all weights in the network one at a time. The feed-forward step is computed in the usual way, but now we also store the output of each unit in its right side. We perform the backpropagation step in the extended network that computes the error function and we then fix our attention on one of the weights, say w_{ij} whose associated edge points from the i-th to the

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

j-th node in the network. This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network. The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i. The backpropagation step computes the gradient of E with respect to this input, i.e., $\partial E/\partial o_i w_{ij}$. Since in the backpropagation step o_i is treated as a constant, we finally have

$$\frac{\partial E}{\partial w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}}.$$

Summarizing, the backpropagation step is performed in the usual way. All subnetworks defined by each weight of the network can be handled simultaneously, but we now store additionally at each node i:

- The output o_i of the node in the feed-forward step.
- The cumulative result of the backward computation in the backpropagation step up to this node. We call this quantity the *backpropagated error*.

If we denote the backpropagated error at the j-th node by δ_j, we can then express the partial derivative of E with respect to w_{ij} as:

$$\frac{\partial E}{\partial w_{ij}} = o_i \delta_j.$$

Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$$\Delta w_{ij} = -\gamma o_i \delta_j.$$

This correction step is needed to transform the backpropagation algorithm into a learning method for neural networks.

This graphical proof of the backpropagation algorithm applies to arbitrary feed-forward topologies. The graphical approach also immediately suggests hardware implementation techniques for backpropagation.

7.3 The case of layered networks

An important special case of feed-forward networks is that of layered networks with one or more hidden layers. In this section we give explicit formulas for the weight updates and show how they can be calculated using linear algebraic operations. We also show how to label each node with the backpropagated error in order to avoid redundant computations.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

7.3.1 Extended network

We will consider a network with n input sites, k hidden, and m output units. The weight between input site i and hidden unit j will be called $w_{ij}^{(1)}$. The weight between hidden unit i and output unit j will be called $w_{ij}^{(2)}$. The bias $-\theta$ of each unit is implemented as the weight of an additional edge. Input vectors are thus extended with a 1 component, and the same is done with the output vector from the hidden layer. Figure 7.17 shows how this is done. The weight between the constant 1 and the hidden unit j is called $w_{n+1,j}^{(1)}$ and the weight between the constant 1 and the output unit j is denoted by $w_{k+1,j}^{(2)}$.

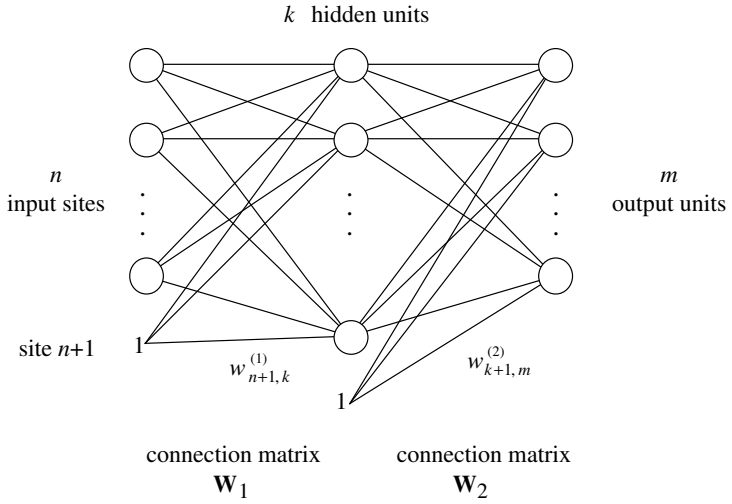

Fig. 7.17. Notation for the three-layered network

There are $(n+1) \times k$ weights between input sites and hidden units and $(k+1) \times m$ between hidden and output units. Let $\overline{\mathbf{W}}_1$ denote the $(n+1) \times k$ matrix with component $w_{ij}^{(1)}$ at the i-th row and the j-th column. Similarly let $\overline{\mathbf{W}}_2$ denote the $(k+1) \times m$ matrix with components $w_{ij}^{(2)}$. We use an overlined notation to emphasize that the last row of both matrices corresponds to the biases of the computing units. The matrix of weights without this last row will be needed in the backpropagation step. The n-dimensional input vector $\mathbf{o} = (o_1, \ldots, o_n)$ is extended, transforming it to $\hat{\mathbf{o}} = (o_1, \ldots, o_n, 1)$. The excitation net_j of the j-th hidden unit is given by

$$net_j = \sum_{i=1}^{n+1} w_{ij}^{(1)} \hat{o}_i.$$

The activation function is a sigmoid and the output $o_j^{(1)}$ of this unit is thus

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$$o_j^{(1)} = s\left(\sum_{i=1}^{n+1} w_{ij}^{(1)} \hat{o}_i\right).$$

The excitation of all units in the hidden layer can be computed with the vector-matrix multiplication $\hat{\mathbf{o}}\overline{\mathbf{W}}_1$. The vector $\mathbf{o}^{(1)}$ whose components are the outputs of the hidden units is given by

$$\mathbf{o}^{(1)} = s(\hat{\mathbf{o}}\overline{\mathbf{W}}_1),$$

using the convention of applying the sigmoid to each component of the argument vector. The excitation of the units in the output layer is computed using the extended vector $\hat{\mathbf{o}}^{(1)} = (o_1^{(1)}, \ldots, o_k^{(1)}, 1)$. The output of the network is the m-dimensional vector $\mathbf{o}^{(2)}$, where

$$\mathbf{o}^{(2)} = s(\hat{\mathbf{o}}^{(1)}\overline{\mathbf{W}}_2).$$

These formulas can be generalized for any number of layers and allow direct computation of the flow of information in the network with simple matrix operations.

7.3.2 Steps of the algorithm

Figure 7.18 shows the extended network for computation of the error function. In order to simplify the discussion we deal with a single input-output pair $(\mathbf{o}, \mathbf{t})$ and generalize later to p training examples. The network has been extended with an additional layer of units. The right sides compute the quadratic deviation $\frac{1}{2}(o_i^{(2)} - t_i)$ for the i-th component of the output vector and the left sides store $(o_i^{(2)} - t_i)$. Each output unit i in the original network computes the sigmoid s and produces the output $o_i^{(2)}$. Addition of the quadratic deviations gives the error E. The error function for p input-output examples can be computed by creating p networks like the one shown, one for each training pair, and adding the outputs of all of them to produce the total error of the training set.

After choosing the weights of the network randomly, the backpropagation algorithm is used to compute the necessary corrections. The algorithm can be decomposed in the following four steps:

i) Feed-forward computation

ii) Backpropagation to the output layer

iii) Backpropagation to the hidden layer

iv) Weight updates

The algorithm is stopped when the value of the error function has become sufficiently small.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

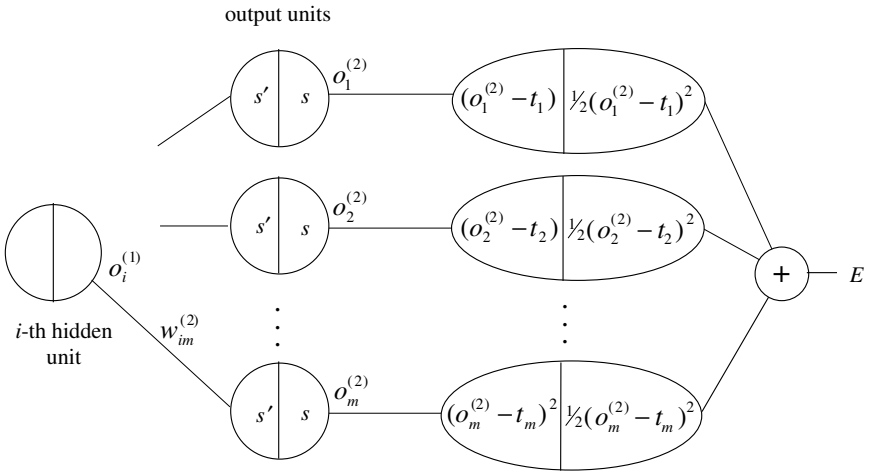

Fig. 7.18. Extended multilayer network for the computation of E

First step: feed-forward computation

The vector $\mathbf{o}$ is presented to the network. The vectors $\mathbf{o}^{(1)}$ and $\mathbf{o}^{(2)}$ are computed and stored. The evaluated derivatives of the activation functions are also stored at each unit.

Second step: backpropagation to the output layer

We are looking for the first set of partial derivatives $\partial E/\partial w_{ij}^{(2)}$. The backpropagation path from the output of the network up to the output unit j is shown in the B-diagram of Figure 7.19.

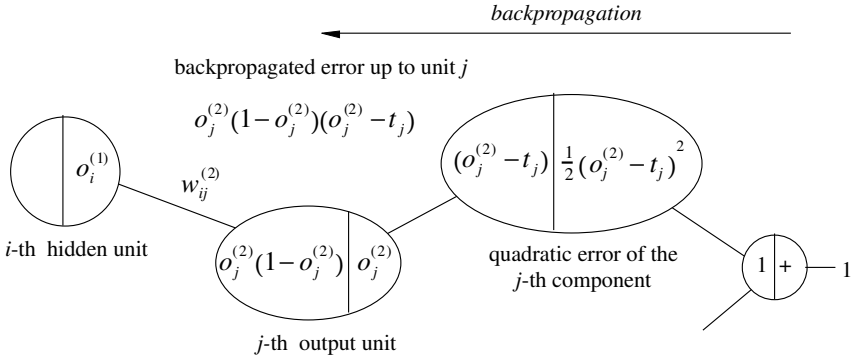

Fig. 7.19. Backpropagation path up to output unit j

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

From this path we can collect by simple inspection all the multiplicative terms which define the backpropagated error $\delta_j^{(2)}$. Therefore

$$\delta_j^{(2)} = o_j^{(2)}(1 - o_j^{(2)})(o_j^{(2)} - t_j),$$

and the partial derivative we are looking for is

$$\frac{\partial E}{\partial w_{ij}^{(2)}} = [o_j^{(2)}(1 - o_j^{(2)})(o_j^{(2)} - t_j)]o_i^{(1)} = \delta_j^{(2)} o_i^{(1)}.$$

Remember that for this last step we consider the weight $w_{ij}^{(2)}$ to be a variable and its input $o_i(1)$ a constant.

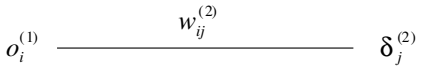

Fig. 7.20. Input and backpropagated error at an edge

Figure 7.20 shows the general situation we find during the backpropagation algorithm. At the input side of the edge with weight w_{ij} we have $o_i^{(1)}$ and at the output side the backpropagated error $\delta_j^{(2)}$.

Third step: backpropagation to the hidden layer

Now we want to compute the partial derivatives $\partial E/\partial w_{ij}^{(1)}$. Each unit j in the hidden layer is connected to each unit q in the output layer with an edge of weight $w_{jq}^{(2)}$, for $q = 1, \ldots, m$. The backpropagated error up to unit j in the hidden layer must be computed taking into account all possible backward paths, as shown in Figure 7.21. The backpropagated error is then

$$\delta_j^{(1)} = o_j^{(1)}(1 - o_j^{(1)}) \sum_{q=1}^{m} w_{jq}^{(2)} \delta_q^{(2)}.$$

Therefore the partial derivative we are looking for is

$$\frac{\partial E}{\partial w_{ij}^{(1)}} = \delta_j^{(1)} o_i.$$

The backpropagated error can be computed in the same way for any number of hidden layers and the expression for the partial derivatives of E keeps the same analytic form.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

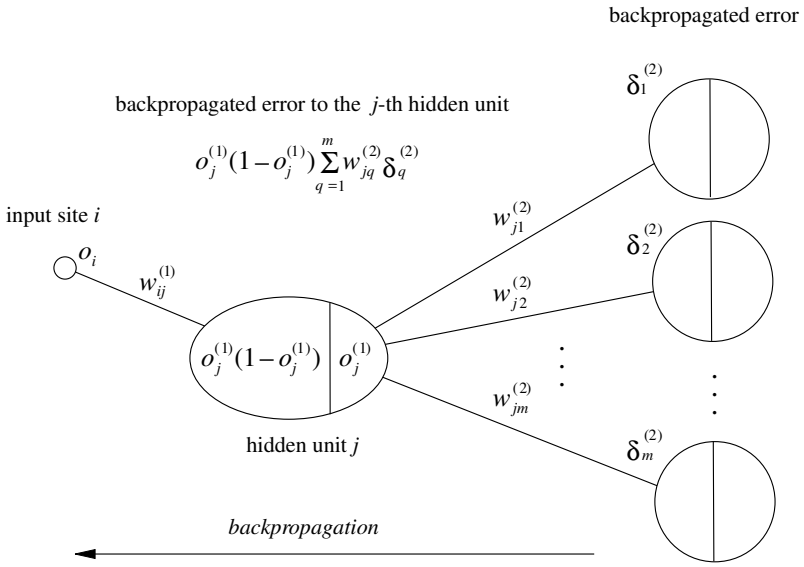

Fig. 7.21. All paths up to input site i

Fourth step: weight updates

After computing all partial derivatives the network weights are updated in the negative gradient direction. A learning constant γ defines the step length of the correction. The corrections for the weights are given by

$$\Delta w_{ij}^{(2)} = -\gamma o_i^{(1)} \delta_j^{(2)}, \quad \text{for } i = 1, \ldots, k+1; j = 1, \ldots, m,$$

and

$$\Delta w_{ij}^{(1)} = -\gamma o_i \delta_j^{(1)}, \quad \text{for } i = 1, \ldots, n+1; j = 1, \ldots, k,$$

where we use the convention that $o_{n+1} = o_{k+1}^{(1)} = 1$. It is very important to make the corrections to the weights only after the backpropagated error has been computed for all units in the network. Otherwise the corrections become intertwined with the backpropagation of the error and the computed corrections do not correspond any more to the negative gradient direction. Some authors fall in this trap [16]. Note also that some books define the backpropagated error as the *negative* traversing value in the network. In that case the update equations for the network weights do not have a negative sign (which is absorbed by the deltas), but this is a matter of pure convention.

More than one training pattern

In the case of $p > 1$ input-output patterns, an extended network is used to compute the error function for each of them separately. The weight corrections

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

are computed for each pattern and so we get, for example, for weight $w_{ij}^{(1)}$ the corrections

$$\Delta_1 w_{ij}^{(1)}, \Delta_2 w_{ij}^{(1)}, \ldots, \Delta_p w_{ij}^{(1)}.$$

The necessary update in the gradient direction is then

$$\Delta w_{ij}^{(1)} = \Delta_1 w_{ij}^{(1)} + \Delta_2 w_{ij}^{(1)} + \cdots + \Delta_p w_{ij}^{(1)}.$$

We speak of *batch* or *off-line* updates when the weight corrections are made in this way. Often, however, the weight updates are made sequentially after each pattern presentation (this is called *on-line* training). In this case the corrections do not exactly follow the negative gradient direction, but if the training patterns are selected randomly the search direction oscillates around the exact gradient direction and, on average, the algorithm implements a form of descent in the error function. The rationale for using on-line training is that adding some noise to the gradient direction can help to avoid falling into shallow local minima of the error function. Also, when the training set consists of thousands of training patterns, it is very expensive to compute the exact gradient direction since each *epoch* (one round of presentation of all patterns to the network) consists of many feed-forward passes and on-line training becomes more efficient [391].

7.3.3 Backpropagation in matrix form

We have seen that the graph labeling approach for the proof of the backpropagation algorithm is completely general and is not limited to the case of regular layered architectures. However this special case can be put into a form suitable for vector processors or special machines for linear algebraic operations.

We have already shown that in a network with a hidden and an output layer (n, k and m units) the input $\mathbf{o}$ produces the output $\mathbf{o}^{(2)} = s(\hat{\mathbf{o}}^{(1)}\overline{\mathbf{W}}_2)$ where $\mathbf{o}^{(1)} = s(\hat{\mathbf{o}}\overline{\mathbf{W}}_1)$. In the backpropagation step we only need the first n rows of matrix $\overline{\mathbf{W}}_1$. We call this $n \times k$ matrix $\mathbf{W}_1$. Similarly, the $k \times m$ matrix $\mathbf{W}_2$ is composed of the first k rows of the matrix $\overline{\mathbf{W}}_2$. We make this reduction because we do not need to backpropagate any values to the constant inputs corresponding to each bias.

The derivatives stored in the feed-forward step at the k hidden units and the m output units can be written as the two diagonal matrices

$$\mathbf{D}_2 = \begin{pmatrix} o_1^{(2)}(1-o_1^{(2)}) & 0 & \cdots & 0 \\ 0 & o_2^{(2)}(1-o_2^{(2)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^{(2)}(1-o_m^{(2)}) \end{pmatrix},$$

and

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$$\mathbf{D}_1 = \begin{pmatrix} o_1^{(1)}(1-o_1^{(1)}) & 0 & \cdots & 0 \\ 0 & o_2^{(1)}(1-o_2^{(1)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_k^{(1)}(1-o_k^{(1)}) \end{pmatrix}.$$

Define the vector $\mathbf{e}$ of the stored derivatives of the quadratic deviations as

$$\mathbf{e} = \begin{pmatrix} (o_1^{(2)} - t_1) \\ (o_2^{(2)} - t_2) \\ \vdots \\ (o_m^{(2)} - t_m) \end{pmatrix}$$

The m-dimensional vector $\boldsymbol{\delta}^{(2)}$ of the backpropagated error up to the output units is given by the expression

$$\boldsymbol{\delta}^{(2)} = \mathbf{D}_2\mathbf{e}.$$

The k-dimensional vector of the backpropagated error up to the hidden layer is

$$\boldsymbol{\delta}^{(1)} = \mathbf{D}_1\mathbf{W}_2\boldsymbol{\delta}^{(2)}.$$

The corrections for the matrices $\overline{\mathbf{W}}_1$ and $\overline{\mathbf{W}}_2$ are then given by

$$\Delta\overline{\mathbf{W}}_2^{\mathrm{T}} = -\gamma\boldsymbol{\delta}^{(2)}\hat{\mathbf{o}}^1 \tag{7.1}$$

and

$$\Delta\overline{\mathbf{W}}_1^{\mathrm{T}} = -\gamma\boldsymbol{\delta}^{(1)}\hat{\mathbf{o}}. \tag{7.2}$$

The only necessary operations are vector-matrix, matrix-vector, and vector-vector multiplications. In Chap. 16 we describe computer architectures optimized for this kind of operation. It is easy to generalize these equations for ℓ layers of computing units. Assume that the connection matrix between layer i and $i+1$ is denoted by $\overline{\mathbf{W}}_{i+1}$ (layer 0 is the layer of input sites). The backpropagated error to the output layer is then

$$\boldsymbol{\delta}^{(\ell)} = \mathbf{D}_\ell\mathbf{e}.$$

The backpropagated error to the i-th computing layer is defined recursively by

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}_i\mathbf{W}_{i+1}\boldsymbol{\delta}^{(i+1)}, \qquad \text{for } i = 1, \ldots, \ell-1.$$

or alternatively

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}_i\mathbf{W}_{i+1}\cdots\mathbf{W}_{\ell-1}\mathbf{D}_{\ell-1}\mathbf{W}_\ell\mathbf{D}_\ell\mathbf{e}.$$

The corrections to the weight matrices are computed in the same way as for two layers of computing units.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

7.3.4 The locality of backpropagation

We can now prove using a B-diagram that addition is the only integration function which preserves the locality of learning when backpropagation is used.

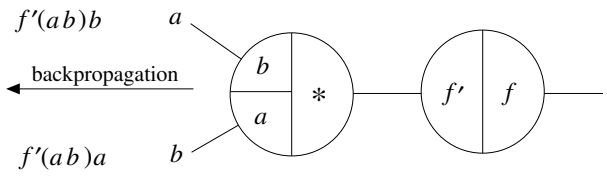

Fig. 7.22. Multiplication as integration function

In the networks we have seen so far the backpropagation algorithm exploits only local information. This means that only information which arrives from an input line in the feed-forward step is stored at the unit and sent back through the same line in the backpropagation step. An example can make this point clear. Assume that the integration function of a unit is multiplication and its activation function is f. Figure 7.22 shows a split view of the computation: two inputs a and b come from two input lines, the integration function responds with the product ab and this result is passed as argument to f. With backpropagation we can compute the partial derivative of $f(ab)$ with respect to a and with respect to b. But in this case the value b must be transported back through the upper edge and the value a through the lower one. Since b arrived through the other edge, the locality of the learning algorithm has been lost. The question is which kinds of integration functions preserve the locality of learning. The answer is given by the following proposition.

Proposition 12. *In a unit with n inputs $x_1, \ldots, x_n$ only integration functions of the form*

$$I(x_1, \ldots, x_n) = F_1(x_1) + F_2(x_2) + \cdots + F_n(x_n) + C,$$

where C is a constant, guarantee the locality of the backpropagation algorithm in the sense that at an edge $i \neq j$ no information about x_j has to be explicitly stored.

Proof. Let the integration function of the unit be the function I of n arguments. If, in the backpropagation step, only a function f_i of the variable x_i can be stored at the computing unit in order to be transmitted through the i-th input line in the backpropagation step, then we know that

$$\frac{\partial I}{\partial x_i} = f_i(x_i), \quad \text{for } i = 1, \ldots, n.$$

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Therefore by integrating these equations we obtain:

$$\begin{aligned} I(x_1, x_2, \ldots, x_n) &= F_1(x_1) + G_1(x_2, \ldots, x_n), \\ I(x_1, x_2, \ldots, x_n) &= F_2(x_2) + G_2(x_2, \ldots, x_n), \\ &\vdots \\ I(x_1, x_2, \ldots, x_n) &= F_n(x_n) + G_n(x_2, \ldots, x_n), \end{aligned}$$

where F_i denotes the integral of f_i and $G_1, \ldots, G_n$ are real functions of $n-1$ arguments. Since the function I has a unique form, the only possibility is

$$I(x_1, x_2, \ldots, x_n) = F_1(x_1) + F_2(x_2) + \cdots + F_n(x_n) + C$$

where C is a constant. This means that information arriving from each line can be preprocessed by the F_i functions and then has to be added. Therefore only integration functions with this form preserve locality. □

7.3.5 Error during training

We discussed the form of the error function for the XOR problem in the last chapter. It is interesting to see how backpropagation performs when confronted with this problem. Figure 7.23 shows the evolution of the total error during training of a network of three computing units. After 600 iterations the algorithm found a solution to the learning problem. In the figure the error falls fast at the beginning and end of training. Between these two zones lies a region in which the error function seems to be almost flat and where progress is slow. This corresponds to a region which would be totally flat if step functions were used as activation functions of the units. Now, using the sigmoid, this region presents a small slope in the direction of the global minimum.

In the next chapter we discuss how to make backpropagation converge faster, taking into account the behavior of the algorithm at the flat spots of the error function.

7.4 Recurrent networks

The backpropagation algorithm can also be extended to the case of recurrent networks. To deal with this kind of systems we introduce a discrete time variable t. At time t all units in the network recompute their outputs, which are then transmitted at time $t+1$. Continuing in this step-by-step fashion, the system produces a sequence of output values when a constant or time varying input is fed into the network. As we already saw in Chap. 2, a recurrent network behaves like a finite automaton. The question now is how to train such an automaton to produce a desired sequence of output values.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

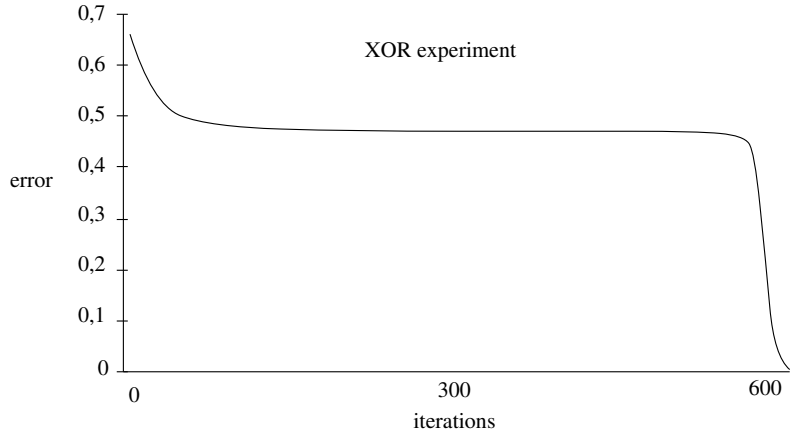

Fig. 7.23. Error function for 600 iterations of backpropagation

7.4.1 Backpropagation through time

The simplest way to deal with a recurrent network is to consider a finite number of iterations only. Assume for generality that a network of n computing units is fully connected and that w_{ij} is the weight associated with the edge from node i to node j. By unfolding the network at the time steps $1, 2, \ldots, T$, we can think of this recurrent network as a feed-forward network with T stages of computation. At each time step t an external input $\mathbf{x}(t)$ is fed into the network and the outputs $(o_1^{(t)}, \ldots, o_n^{(t)})$ of all computing units are recorded. We call the n-dimensional vector of the units' outputs at time t the network state $\mathbf{o}^{(t)}$. We assume that the initial values of all unit's outputs are zero at $t = 0$, but the external input $\mathbf{x}(0)$ can be different from zero. Figure 7.24 shows a diagram of the unfolded network. This unfolding strategy which converts a recurrent network into a feed-forward network in order to apply the backpropagation algorithm is called *backpropagation through time* or just BPTT [383].

Let $\mathbf{W}$ stand for the $n \times n$ matrix of network weights w_{ij}. Let $\mathbf{W}_0$ stand for the $m \times n$ matrix of interconnections between m input sites and n units. The feed-forward step is computed in the usual manner, starting with an initial m-dimensional external input $\mathbf{x}^{(0)}$. At each time step t the network state $\mathbf{o}^{(t)}$ (an n-dimensional row vector) and the vector of derivatives of the activation function at each node $\mathbf{o}'^{(t)}$ are stored. The error of the network can be measured after each time step if a sequence of values is to be produced, or just after the final step T if only the final output is of importance. We will handle the first, more general case. Denote the difference between the n-dimensional target $\mathbf{y}^{(t)}$ at time t and the output of the network by $\mathbf{e}^{(t)} = \left(\mathbf{o}^{(t)} - \mathbf{y}^{(t)}\right)^{\mathrm{T}}$. This is an n-dimensional column vector, but in most cases we are only interested in the outputs of some units in the network. In that case

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

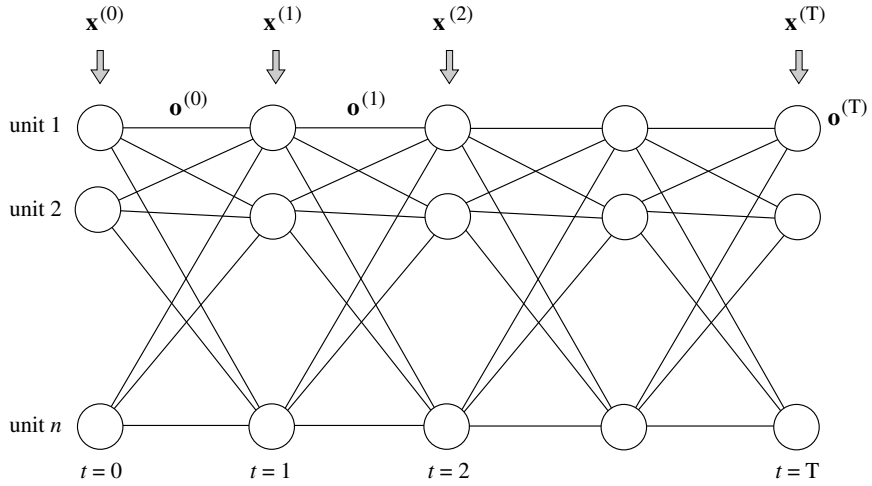

Fig. 7.24. Backpropagation through time

define $e_i(t) = 0$ for each unit i, whose precise state is unimportant and which can remain hidden from view.

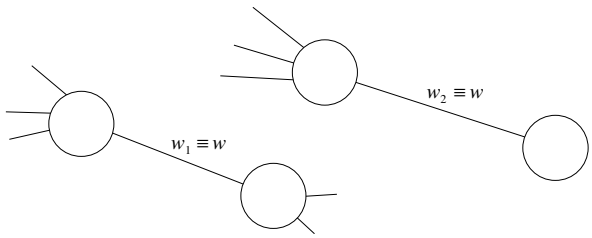

Fig. 7.25. A duplicated weight in a network

Things become complicated when we consider that each weight in the network is present at each stage of the unfolded network. Until now we had only handled the case of unique weights. However, any network with repeated weights can easily be transformed into a network with unique weights. Assume that after the feed-forward step the state of the network is the one shown in Figure 7.25. Weight w is duplicated, but received different inputs o_1 and o_2 in the feed-forward step at the two different locations in the network. The transformed network in Figure 7.26 is indistinguishable from the original network from the viewpoint of the results it produces. Note that the two edges associated with weight w now have weight 1 and a multiplication is performed by the two additional units in the middle of the edges. In this transformed network w appears only once and we can perform backpropagation as usual. There are two groups of paths, the ones coming from the first multiplier to w

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

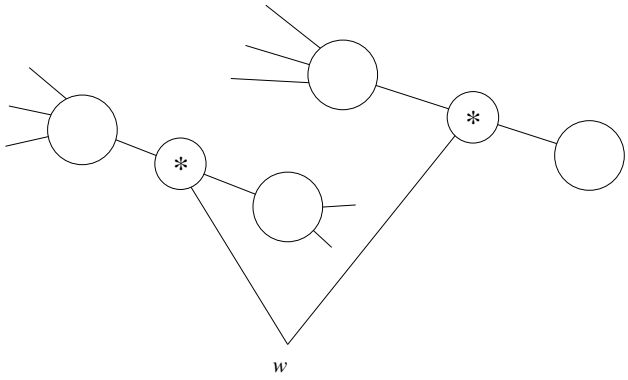

Fig. 7.26. Transformed network

and the ones coming from the second. This means that we can just perform backpropagation as usual in the original network. At the first edge we obtain $\partial E/\partial w_1$, at the second $\partial E/\partial w_2$, and since w_1 is the same variable as w_2, the desired partial derivative is

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}.$$

We can thus conclude in general that in the case of the same weight being associated with several edges, backpropagation is performed as usual for each of those edges and the results are simply added.

The backpropagation phase of BPTT starts from the right. The backpropagated error at time T is given by

$$\boldsymbol{\delta}^{(T)} = \mathbf{D}^{(T)}\mathbf{e}^{(T)},$$

where $\mathbf{D}^{(T)}$ is the $n \times n$ diagonal matrix whose component at the i-th diagonal element is $o_i'^{(T)}$, i.e., the stored derivative of the i-th unit output at time T. The backpropagated error at time $T-1$ is given by

$$\boldsymbol{\delta}^{(T-1)} = \mathbf{D}^{(T-1)}\mathbf{e}^{(T-1)} + \mathbf{D}^{(T-1)}\mathbf{W}\mathbf{D}^{(T)}\mathbf{e}^{(T)},$$

where we have considered all paths from the computed errors at time T and $T-1$ to each weight. In general, the backpropagated error at stage i, for $i = 0, \ldots, T-1$ is

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}^{(i)}(\mathbf{e}^{(i)} + \mathbf{W}\boldsymbol{\delta}^{(i+1)}).$$

The analytic expression for the final weight corrections are

$$\Delta\overline{\mathbf{W}}^{T} = -\gamma\left(\boldsymbol{\delta}^{(1)}\hat{\mathbf{o}}^{(0)} + \cdots + \boldsymbol{\delta}^{(T)}\hat{\mathbf{o}}^{(T-1)}\right) \tag{7.3}$$

$$\Delta\overline{\mathbf{W}}_0^{T} = -\gamma\left(\boldsymbol{\delta}^{(0)}\hat{\mathbf{x}}^{(0)} + \cdots + \boldsymbol{\delta}^{(T)}\hat{\mathbf{x}}^{(T)}\right), \tag{7.4}$$

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

where $\hat{\mathbf{o}}^{(1)}, \ldots, \hat{\mathbf{o}}^{(T)}$ denote the extended output vectors at steps $1, \ldots, T$ and $\overline{\mathbf{W}}$ and $\overline{\mathbf{W}}_0$ the extended matrices $\mathbf{W}$ and $\mathbf{W}_0$.

Backpropagation through time can be extended to the case of infinite time T. In this case we are interested in the limit value of the network's state, on the assumption that the network's state stabilizes to a fixpoint $\tilde{\mathbf{o}}$. Under some conditions over the activation functions and network topology such a fixpoint exists and its derivative can be calculated by backpropagation. The feed-forward step is repeated a certain number of times, until the network relaxes to a numerically stable state (with certain precision). The stored node's outputs and derivatives are the ones computed in the last iteration. The network is then run backwards in backpropagation manner until it reaches a numerically stable state. The gradient of the network function with respect to each weight can then be calculated in the usual manner. Note that in this case, we do not need to store all intermediate values of outputs and derivatives at the units, only the final ones. This algorithm, called recurrent backpropagation, was proposed independently by Pineda [342] and Almeida [20].

7.4.2 Hidden Markov Models

Hidden Markov Models (HMM) form an important special type of recurrent network. A first-order Markov model is any system capable of assuming one of n different states at time t. The system does not change its state at each time step deterministically but according to a stochastic dynamics. The probability of transition from the i-th to the j-th state at each step is given by $0 \leq a_{ij} \leq 1$ and does not depend on the previous history of transitions. These probabilities can be arranged in an $n \times n$ matrix A. We also assume that at each step the model emits one of m possible output values. We call the probability of emitting the k-th output value while in the i-th state b_{ik}. Starting from a definite state at time $t = 0$, the system is allowed to run for T time units and the generated outputs are recorded. Each new run of the system generally produces a different sequence of output values. The system is called a HMM because only the emitted values, not the state transitions, can be observed.

An example may make this point clear. In speech recognition researchers postulate that the vocal tract shapes can be quantized in a discrete set of states roughly associated with the phonemes which compose speech. When speech is recorded the exact transitions in the vocal tract cannot be observed and only the produced sound can be measured at some predefined time intervals. These are the emissions, and the states of the system are the quantized configurations of the vocal tract. From the measurements we want to infer the sequence of states of the vocal tract, i.e., the sequence of utterances which gave rise to the recorded sounds. In order to make this problem manageable, the set of states and the set of possible sound parameters are quantized (see Chap. 9 for a deeper discussion of automatic speech recognition).

The general problem when confronted with the recorded sequence of output values of a HMM is to compute the most probable sequence of state

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

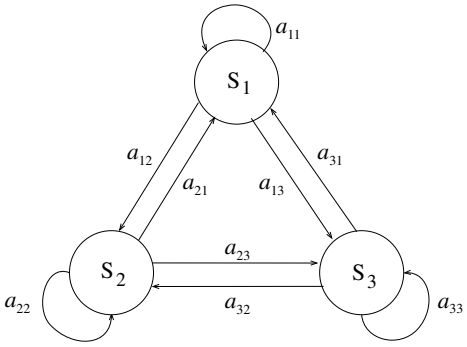

Fig. 7.27. Transition probabilities of a Markov model with three states

transitions which could have produced them. This is done with a recursive algorithm.

The state diagram of a HMM can be represented by a network made of n units (one for each state) and with connections from each unit to each other. The weights of the connections are the transition probabilities (Figure 7.27).

As in the case of backpropagation through time, we can unfold the network in order to observe it at each time step. At $t = 0$ only one of the n units, say the i-th, produces the output 1, all others zero. State i is the the actual state of the system. The probability that at time $t = 1$ the system reaches state j is given by a_{ij} (to avoid cluttering only some of these values are shown in the diagram). The probability of reaching state k at $t = 2$ is

$$\sum_{j=1}^{n} a_{ij} a_{jk}$$

which is just the net input at the k-th node in the stage $t = 2$ of the network shown in Figure 7.28. Consider now what happens when we can only observe the output of the system but not the state transitions (refer to Figure 7.29). If the system starts at $t = 0$ in a state given by a discrete probability distribution $\rho_1, \rho_2, \ldots, \rho_n$, then the probability of observing the k-th output at $t = 0$ is given by

$$\sum_{i=1}^{n} \rho_i b_{ik}.$$

The probability of observing the k-th output at $t = 0$ *and* the m-th output at $t = 1$ is

$$\sum_{j=1}^{n} \sum_{i=1}^{n} \rho_i b_{ik} a_{ij} b_{jm}.$$

The rest of the stages of the network compute the corresponding probabilities in a similar manner.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

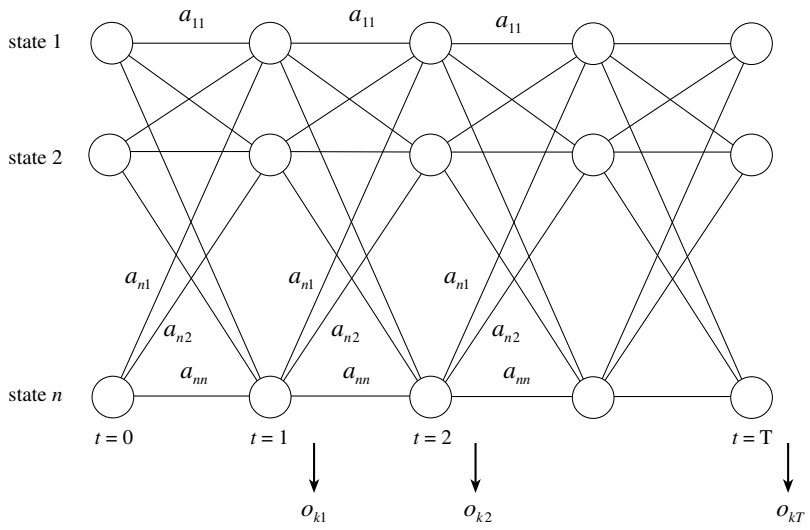

Fig. 7.28. Unfolded Hidden Markov Model

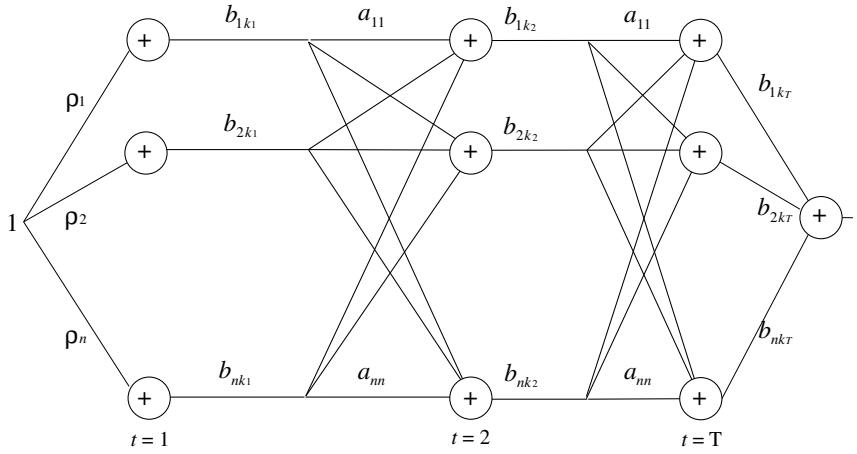

Fig. 7.29. Computation of the likelihood of a sequence of observations

How can we find the unknown transition and emission probabilities for such an HMM? If we are given a sequence of T observed outputs with indices $k_1, k_2, \ldots, k_T$ we would like to maximize the likelihood of this sequence, i.e., the product of the probabilities that each of them occurs. This can be done by transforming the unfolded network as shown in Figure 7.29 for $T = 3$. Notice that at each stage h we introduced an additional edge from the node i with the weight b_{i,k_h}. In this way the final node which collects the sum of the whole computation effectively computes the likelihood of the observed sequence.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Since this unfolded network contains only differentiable functions at its nodes (in fact only addition and the identity function) it can be trained using the backpropagation algorithm. However, care must be taken to avoid updating the probabilities in such a way that they become negative or greater than 1. Also the transition probabilities starting from the same node must always add up to 1. These conditions can be enforced by an additional transformation of the network (introducing for example a "softmax" function [39]) or using the method of Lagrange multipliers. We give only a hint of how this last technique can be implemented so that the reader may complete the network by her- or himself.

Assume that a function F of n parameters $x_1, x_2, \ldots, x_n$ is to be minimized, subject to the constraint $C(x_1, x_2, \ldots, x_n) = 0$. We introduce a Lagrange multiplier λ and define the new function

$$L(x_1, \ldots, x_n, \lambda) = F(x_1, \ldots, x_n) + \lambda C(x_1, \ldots, x_n).$$

To minimize L we compute its gradient and set it to zero. To do this numerically, we follow the negative gradient direction to find the minimum. Note that since

$$\frac{\partial L}{\partial \lambda} = C(x_1, \ldots, x_n)$$

the iteration process does not finish as long as $C(x_1, \ldots, x_n) \neq 0$, because in that case the partial derivative of L with respect to λ is non-zero. If the iteration process converges, we can be sure that the constraint C is satisfied. Care must be taken when the minimum of F is reached at a saddle point of L. In this case some modifications of the basic gradient descent algorithm are needed [343]. Figure 7.30 shows a diagram of the network (a *Lagrange neural network* [468]) adapted to include a constraint. Since all functions in the network are differentiable, the partial derivatives needed can be computed with the backpropagation algorithm.

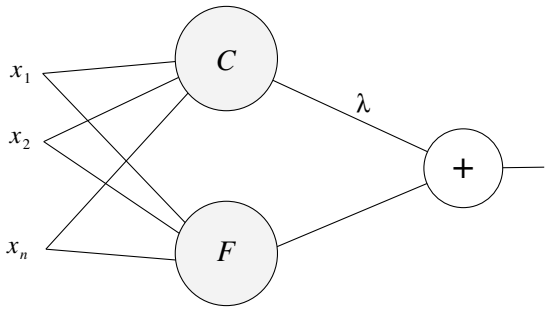

Fig. 7.30. Lagrange neural network

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

7.4.3 Variational problems

Our next example, deals not with a recurrent network, but with a class of networks built of many repeated stages. Variational problems can also be expressed and solved numerically using backpropagation networks. A variational problem is one in which we are looking for a function which can optimize a certain cost function. Usually cost is expressed analytically in terms of the unknown function and finding a solution is in many cases an extremely difficult problem. An example can illustrate the general technique that can be used.

Assume that the problem is to minimize P with two boundary conditions:

$$P(u) = \int_o^1 F(u, u')dx \qquad \text{with } u(0) = a \quad \text{and} \quad u(1) = b.$$

Here u is an unknown function of x and $F(u, u')$ a cost function. P represents the total cost associated with u over the interval $[0, 1]$. Since we want to solve this problem numerically, we discretize the function u by partitioning the interval $[0, 1]$ into $n - 1$ subintervals. The discrete successive values of the function are denoted by $u_1, u_2, \ldots, u_n$, where $u_1 = a$ and $u_n = b$ are the boundary conditions. The length of the subintervals is $\Delta x = 1/(n - 1)$. The discrete function P_d that we want to minimize is thus:

$$P_d(u) = \sum_{i=1}^{n} F(u_i, u_i')\Delta x.$$

Since minimizing $P_d(u)$ is equivalent to minimizing $P_D(u) = P_d(u)/\Delta x$ (Δx is constant), we proceed to minimize $P_D(u)$. We can approximate the derivative u_i' by $(u_i - u_{i-1})/\Delta x$. Figure 7.31 shows the network that computes the discrete approximation $P_D(u)$.

We can now compute the gradient of P_D with respect to each u_i by performing backpropagation on this network. Note that there are three possible paths from P_D to u_i, so the partial derivative of P_D with respect to u_i is

$$\frac{\partial P_D}{\partial u_i} = \underbrace{\frac{\partial F}{\partial u}(u_i, u_i')}_{path1} + \underbrace{\frac{1}{\Delta x}\frac{\partial F}{\partial u'}(u_i, u_i')}_{path2} - \underbrace{\frac{1}{\Delta x}\frac{\partial F}{\partial u'}(u_{i+1}, u_{i+1}')}_{path3}$$

which can be rearranged to

$$\frac{\partial P_d}{\partial u_i} = \frac{\partial F}{\partial u}(u_i, u_i') - \frac{1}{\Delta x}\left(\frac{\partial F}{\partial u'}(u_{i+1}, u_{i+1}') - \frac{\partial F}{\partial u'}(u_i, u_i')\right).$$

At the minimum all these terms should vanish and we get the expression

$$\frac{\partial F}{\partial u}(u_i, u_i') - \frac{1}{\Delta x}\left(\frac{\partial F}{\partial u'}(u_{i+1}, u_{i+1}') - \frac{\partial F}{\partial u'}(u_i, u_i')\right) = 0$$

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

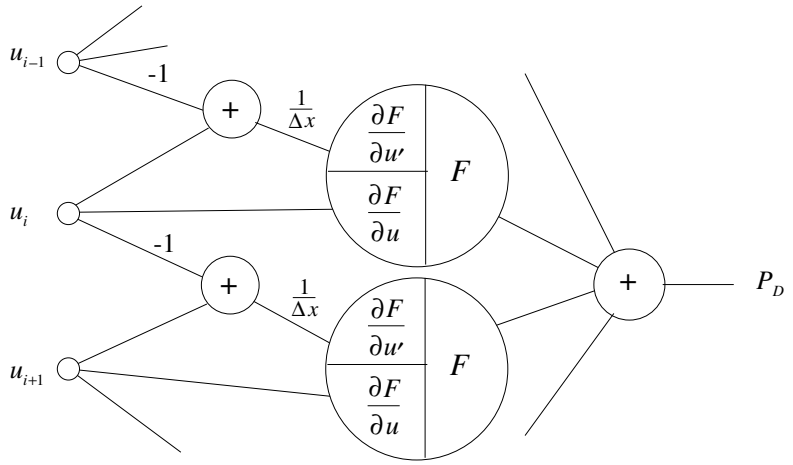

Fig. 7.31. A network for variational calculus

which is the discrete version of the celebrated Euler equation

$$\frac{\partial F}{\partial u} - \frac{d}{dx}\left(\frac{\partial F}{\partial u'}\right) = 0.$$

In fact this can be considered a simple derivation of Euler's result in a discrete setting.

By selecting another function F many variational problems can be solved numerically. The curve of minimal length between two points can be found by using the function

$$F(u, u') = \sqrt{1 + u'^2} = \frac{\sqrt{dx^2 + du^2}}{dx}$$

which when integrated with respect to x corresponds to the path length between the boundary conditions. In 1962 Dreyfus solved the constrained brachystochrone problem, one of the most famous problems of variational calculus, using a numerical approach similar to the one discussed here [115].

7.5 Historical and bibliographical remarks

The field of neural networks started with the investigations of researchers of the caliber of McCulloch, Wiener, and von Neumann. The perceptron era was its *Sturm und Drang* period, the epoch in which many new ideas were tested and novel problems were being solved using perceptrons. However, at the end of the 1960s it became evident that more complex multilayered architectures demanded a new learning paradigm. In the absence of such an algorithm,

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

a new era of cautious incremental improvements and silent experimentation began.

The algorithm that the neural network community needed had already been developed by researchers working in the field of optimal control. These researchers were dealing with variational problems with boundary conditions in which a function capable of optimizing a cost function subject to some constraints must be determined. As in the field of neural networks, a function f must be found and a set of input-output values is predefined for some points. Around 1960 Kelley and Bryson developed numerical methods to solve this variational problem which relied on a recursive calculation of the gradient of a cost function according to its unknown parameters [241, 76]. In 1962 Dreyfus, also known for his criticism of symbolic AI, showed how to express the variational problems as a kind of multistage system and gave a simple derivation of what we now call the backpropagation algorithm [115, 116]. He was the first to use an approach based on the chain rule, in fact one very similar to that used later by the PDP group. Bryson and Ho later summarized this technique in their classical book on optimal control [76]. However, Bryson gives credit for the idea of using numerical methods to solve variational problems to Courant, who in 1943 proposed using gradient descent along the Euler expression (the partial derivative of the cost function) to find numerical approximations to variational problems [93].

The algorithm was redeveloped by some other researchers working in the field of statistics or pattern recognition. We can look as far back as Gauss to find mathematicians already doing function-fitting with numerical methods. Gauss developed the method of least squares and considered the fitting of nonlinear functions of unknown parameters. In the Gauss–Newton method the function F of parameters $w_1, \ldots, w_n$ is approximated by its Taylor expansion at an initial point using only the first-order partial derivatives. Then a least-squares problem is solved and a new set of parameters is found. This is done iteratively until the function F approximates the empirical data with the desired accuracy [180]. Another possibility, however, is the use of the partial derivatives of F with respect to the parameters to do a search in the gradient direction. This approach was already being used by statisticians in the 1960s [292]. In 1974 Werbos considered the case of general function composition and proposed the backpropagation algorithm [442, 443] as a kind of nonlinear regression. The points given as the training set are considered not as boundary conditions, which cannot be violated, but as experimental points which have to be approximated by a suitable function. The special case of recursive backpropagation for Hidden Markov Models was solved by Baum, also considering the restrictions on the range of probability values [47], which he solved by doing a projective transformation after each update of the set of probabilities. His "forward-backward" algorithm for HMMs can be considered one of the precursors of the backpropagation algorithm.

Finally, the AI community also came to rediscovering backpropagation on its own. Rumelhart and his coauthors [383] used it to optimize multilayered

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

neural networks in the 1980s. Le Cun is also mentioned frequently as one of the authors who reinvented backpropagation [269]. The main difference however to the approach of both the control or statistics community was in conceiving the networks of functions as interconnected computing units. We said before that backpropagation reduces to the recursive computation of the chain rule. But there is also a difference: the network of computing units serves as the underlying data structure to store values of previous computations, avoiding redundant work by the simple expedient of running the network backwards and labeling the nodes with the backpropagated error. In this sense the backpropagation algorithm, as rediscovered by the AI community, added something new, namely the concept of functions as dynamical objects being evaluated by a network of computing elements and backpropagation as an inversion of the network dynamics.

Exercises

1. Implement the backpropagation algorithm and train a network that computes the parity of 5 input bits.
2. The symmetric sigmoid is defined as $t(x) = 2s(x) - 1$, where $s(\cdot)$ is the usual sigmoid function. Find the new expressions for the weight corrections in a layered network in which the nodes use t as a primitive function.
3. Find the analytic expressions for the partial derivative of the error function according to each one of the input values to the network. What could be done with this kind of information?
4. Find a discrete approximation to the curve of minimal length between two points in $\mathbb{R}^3$ using a backpropagation network.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996