Back Propagation

J. Henseler[*]

Forensic Science Laboratory of the Ministry of Justice, Rijswijk

1 Introduction

In the late 1950's two artificial neural networks were introduced that have had a great impact on current neural network models. The first one is known as the *Perceptron* (cf. Rosenblatt, 1958, 1962) and contains linear threshold units, i.e., outputs are either zero or one. The second network model is constructed from Adaline (*Ada*ptive *Line*ar) units which have a linear output, i.e., without a threshold. This network is known as the *Madaline* (cf. Widrow and Hoff, 1960). Both networks use a learning rule that is a variant of what is now called the *delta rule* (Rumelhart et al., 1986).

The main drawback of these two neural network models is their restriction to one layer of adaptive connections. In their famous book *Perceptrons*, Minsky and Papert (1969) showed that such networks are only capable of associating linearly separable input classes. This means, for example, that neither the Perceptron nor the Madaline would ever be able to learn the exclusive-or (XOR) problem (cf. Section 2). Minsky and Papert also noted that these limitations could be overcome if an intermediate layer of adaptive connections is introduced. At that time, however, no efficient learning rule for networks with intermediate layers was known.

In 1985 several learning schemes for adapting intermediate connections were reported (Parker, 1985; Le Cun, 1985). However, in this paper we will focus entirely on the *generalized delta rule* that was introduced in 1986 by Rumelhart et al. (1986). The application of the generalized delta rule requires two phases. In the first phase, input is propagated forward to the output units where the error of the network is measured. In the second phase, the error is propagated backward through the network and is used for adapting connections. Owing to the second phase this procedure is also known as *Back Propagation* of error. We note that this procedure is similar to an algorithm descibed much earlier by Werbos (1974).

Section 2 describes the Perceptron learning rule. It also shows why a perceptron can not solve the exclusive-or problem. In Section 3 the Madaline learning rule is described and its relation to the Perceptron learning rule. In Section 4 the architecture of multi-layer neural networks is introduced. It describes how these networks may be adapted using the generalized delta rule. Section 5 pays attention to a serious drawback of this learning rule, viz., the existence of local minima. In Section 6 some second-order improvements on the generalized

[*] We thank IBM for their hardware support under the Joint Study Agreement DAEDALOS

delta rule are presented, viz., the ***momentum*** and ***adaptive back propagation***. In Section 7 a recurrent network is described that can be trained with Back Propagation. Such a network may be used for learning patterns with a temporal extent. Finally, in Section 8 an application of a multi-layer neural network for controlling a robot arm is discussed. There are two appendices to this paper. In Appendix A a vectorized version of the generalized delta rule is derived. Appendix B contains a pseudo-code description of the Back Propagation algorithm.

2 Perceptron learning rule

The Perceptron was introduced as a layer of neurons that receive input from a retina. The neurons are ***not interconnected*** and can only be activated by input from the retina. A neuron receives activation from a retina point if and only if (1) there exists a connection, and (2) the retina point itself is activated, e.g., black. The sum of this activation in a neuron is called nett input σ. Neurons in a perceptron are threshold units, i.e., the output of a neuron y is 1 if σ exceeds the threshold θ and 0 if not. Figure 1 depicts a perceptron with inputs $x_1, \ldots, x_n$.

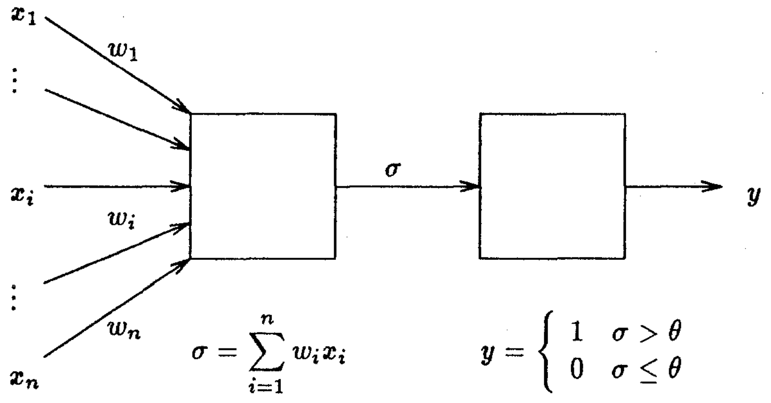
x_1
w_1
x_i
w_i
w_n
x_n
σ
y
$\sigma = \sum_{i=1}^{n} w_i x_i$
$y = \begin{cases} 1 & \sigma > \theta \\ 0 & \sigma \leq \theta \end{cases}$

Fig. 1. Block diagram of a Perceptron neuron processing model.

The processing model of a neuron in a perceptron with inputs $x_1, \ldots, x_n \in R$ and connection weights $w_1, \ldots, w_n \in R$ can mathematically be described as follows:

$$\sigma = \sum_{i=1}^{n} w_i x_i$$
$$y = \begin{cases} 1 & \sigma > \theta \\ 0 & \sigma \leq \theta \end{cases} \tag{1}$$

Here $w_i \neq 0$ means a connection to the i-th input exists and $w_i = 0$ means it does not exist. If $w_i > 0$ the input contributes to the activation sum σ in a positive way, i.e., it *excites* the neuron. If $w_i < 0$ the input decreases the activation sum, i.e., it *inhibits* the neuron. A neuron will only turn on in case the excitatory input is more than θ units stronger than the inhibitory input. Hence, a neuron, or a number of neurons, establishes a mapping from input activity on the output. The nature of this mapping is entirely determined by the perceptron *configuration*, i.e., by connections and thresholds.

In terms of a Perceptron, *pattern recognition* may be interpreted as a mapping of retina images onto a number of categories. A perceptron can perform this task if it has an appropriate configuration. If such a configuration exists then it can be obtained by adapting the perceptron using the procedure presented in Table 1. According to this procedure the connections and thresholds are adapted, based on the actual output of a neuron y (cf. Equation 1) and its desired output, or target, Y.

Table 1. Perceptron Learning Procedure

```
INPUT:  $x_1, \ldots, x_n$
TARGET: $Y$

Calculate $y$ according to Equation 1
if $y \neq Y$ then
    if $y = 1$ then
        $\theta = \theta + 1$
        for $i = 1$ to $n$ do
            if $x_i = 1$ then $w_i = w_i - 1$
        endfor
    else
        $\theta = \theta - 1$
        for $i = 1$ to $n$ do
            if $x_i = 1$ then $w_i = w_i + 1$
        endfor
    endif
endif
```

The perceptron learning procedure can be described as a *delta rule* in mathematical form. Let the threshold change and the weight change be denoted by $\Delta\theta$ and Δw_i, respectively. It is easy to see that the delta rule presented in (2) corresponds to the procedure described in Table 1 :

$$\begin{aligned} \Delta\theta &= y - Y = \delta \\ \Delta w_i &= -(y - Y)x_i = -\delta x_i \end{aligned} \tag{2}$$

For some mappings, however, an appropriate configuration does not exist. As was pointed out by Minsky and Papert (1969) no perceptron configuration can be found for the XOR-function in Table 2. This configuration is known as the exclusive-or problem since either the first input *or* the second input must be activated but *not* both in order to turn on the output. In the XOR case this

Table 2. Mappings of the OR, AND and XOR function, respectively.

Input		Output		
x_1	x_2	OR	AND	XOR
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

problem can be analyzed as follows. A single neuron can only categorize inputs x_1 and x_2 in two classes, viz., Class 0 containing inputs for which $w_1x_1+w_2x_2 \leq \theta$ and Class 1 containing inputs for which $w_1x_1 + w_2x_2 > \theta$. For any value of w_1, w_2 and θ this separation has the shape of a line meaning that Class 0 and 1 have to be *linearly separable*. From Figure 2 it follows that the AND and OR functions are linearly separable but that the XOR function requires a non-linear separation.

If, on the other hand, we were allowed to add another input feature x_3 to the perceptron that is the logical-and function of x_1 and x_2 it would be possible to solve the XOR problem. This input feature could be calculated by a second, intermediate, neuron. In that case, however, the perceptron learning procedure does not tell us how to configure this neuron since no target for its output is known since it is *hidden* from the network output. In Section 4 a generalization of the delta-rule will be presented that is capable of configuring intermediate, or hidden, neurons. First, we will deal with adapting continuous-valued weights in the next section.

3 Gradient descent

The Perceptron Learning Procedure described in Table 1 only works with discrete-valued connections, inputs and outputs. In the Madaline (Widrow and Hoff,

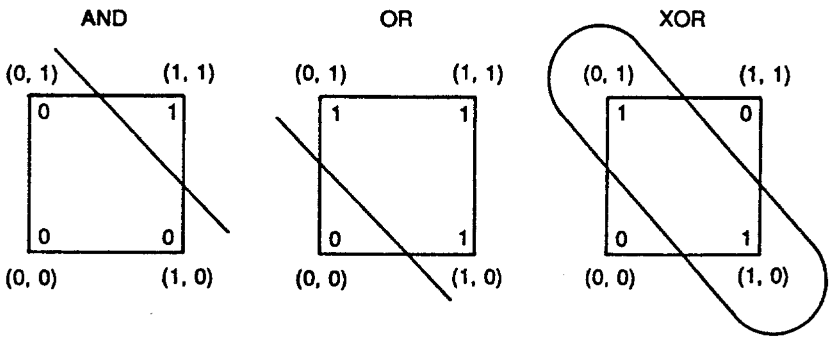

Fig. 2. Geometric representation of the AND, OR and XOR functions.

1960), however, connections are continuous as are the outputs since a linear neuron function is used. Hence, a new learning procedure is required that is capable of minimizing the error for continuous values. In the Madaline this problem was solved by applying a *Least Mean Squares* (LMS) procedure. This approach requires that the error of the system is measured as the sum of the squared errors. For a single target Y the squared-error E is :

$$E = (y - Y)^2 \tag{3}$$

The LMS procedure calculates how the weights needed to produce y should be changed in order to decrease E. Obviously, a correct configuration has been learned if and only if $E = 0$. After this adaptation y is calculated again and the process is repeated. In case more than one pattern must be learned E is calculated as the sum of all pattern-errors. The weights are adapted by cycling through the pattern set and for each pattern adapting the weights according to the individual pattern errors.

The method used for finding the correct adaptation vector $(\Delta w_1, \ldots, \Delta w_n)$ is known as *gradient descent*. If we think of E as a function of $w = (w_1, \ldots, w_n)$, then the gradient of E with respect to w denotes the slope of the "error-surface". By *descending* this surface downhill, i.e., in the direction of the negative *gradient*, we will finally reach at the bottom of the surface. At that point the error can no longer be decreased and the procedure finishes. In section 5 an example of gradient descent for a network with two weights is presented.

The gradient of the error surface can only be calculated if the neuron-processing function is differentiable. Hence, the processing model in Equation 1 can not be used because the output function has an infinite gradient if σ approaches θ. In the Adaline neuron-processing model the threshold is eliminated and a linear function remains, i.e., $y = \sum_{i=1}^{n} w_i x_i$. The corresponding error surface is smooth and the LMS procedure is applicable.

Figure 3 depicts a typical error surface corresponding to the OR-mapping (cf. Table 2) for a single neuron with two weights. Using the linear neuron function the error function $E(w_1, w_2)$ can be written as the sum of the squared errors for each entry in the OR table.

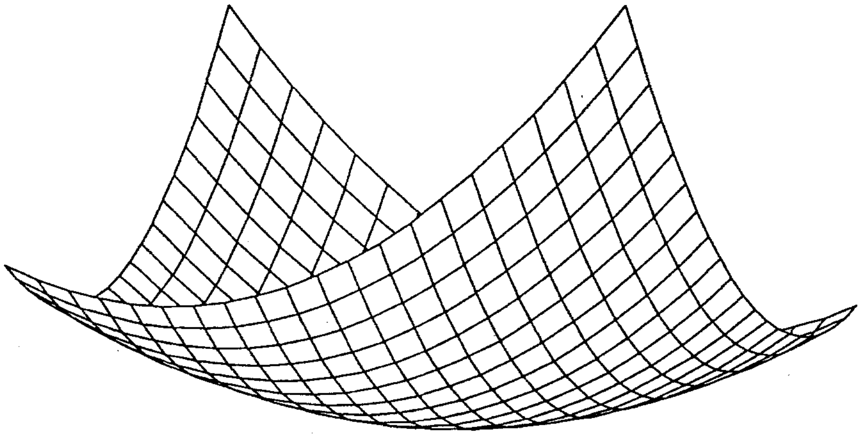

Fig. 3. "Bowl-shaped" error-surface with a minimum at the center.

$$E(w_1, w_2) = (1 - w_1)^2 + (1 - w_2)^2 + (1 - w_1 - w_2)^2 \tag{4}$$

In this case a gradient descent will finally lead to the minimum of E. For the linear neuron model the learning rule for adapting w_i becomes :

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{5}$$

Constant η is called the *learning rate* and determines how much the surface will be descended in one step. Taking large steps, i.e., using a large learning rate, speeds up the learning process. In some cases, however, it may lead to unstable behaviour of the system, e.g. introducing oscillations. By substituting E in equation (5) using Equation 3 and subsequently substituting $y = \sum_{i=1}^{n} w_i x_i$, the derivative of the error measure with respect to w_i (cf. Equation 5) is :

$$\Delta w_i = -\eta \frac{\partial (y - Y)^2}{\partial w_i} = -2\eta (y - Y) \frac{\partial y}{\partial w_i} = -2\eta\delta \frac{\partial \sum_{k=1}^{n} w_k x_k}{\partial w_i} = -2\eta\delta x_i \tag{6}$$

This result is similar to the mathematical form of the Perceptron Learning Rule (cf. Equation 2). We conclude with noting that the LMS procedure is a useful alternative for the Perceptron Learning Procedure described in Table 1 for adapting continuous-valued weights.

4 Back Propagation

In Section 2 it was explained why a Perceptron can not solve the XOR problem, unless a hidden neuron is used. However, since no target output for a such a neuron is specified, neither the Perceptron Learning procedure (cf. Table 1) nor the LMS adaptation for the Madaline (cf. Equation 6) is applicable. Hence, a correct configuration can not be found. The *generalized delta rule* eliminates this problem by using the error gradient of the LMS procedure as a substitute target error for hidden neurons.

4.1 Multi-layer Network

A *multi-layer network* is a special case of a Perceptron with hidden neurons. It consists of a number of consecutive layers, i.e., an input neuron layer, zero or more hidden layers and an output layer. In case there are no hidden layers the multi-layer network is equivalent to a Perceptron; that is, neurons in the same layer are not interconnected and neurons in the input layer represent input features, e.g., pixels. The output of the input layer is presented to the first hidden layer, or, if there are no hidden layers, directly to the output layer. Neurons in a hidden layer that do not receive inputs from the input layer are connected to the neurons in the previous hidden layer. Hence, the output of a hidden neuron is sent to the next layer which may either be another hidden layer or the output layer. Finally, the output layer sends its output to the environment. A multi-layer network consisting of N layers is depicted in Figure 4. We have denoted the number of neurons in layer p by m_p.

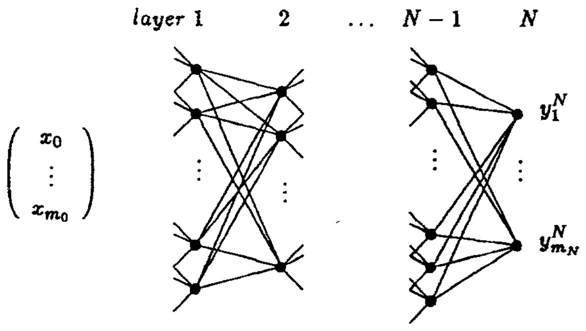

Fig. 4. A multi-layer network consisting of N layers.

Just like in the Madaline, the connections have continuous-valued weights and the neuron input and output are also continuous valued. The connection to the i-th neuron in layer p from the j-th neuron in layer $p-1$ has a weight denoted by w_{ij}^p. Two connected layers $p-1$ and p with their connections and corresponding weights are shown in Figure 5.

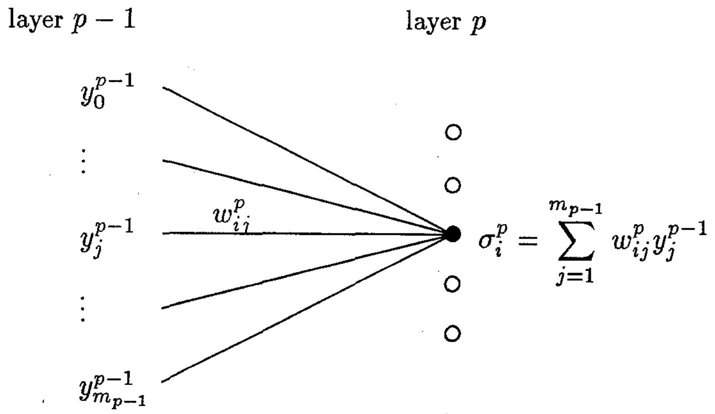

Fig. 5. Organization of connection weights corresponding to a neuron.

We note that if a multi-layer network would be constructed from Adalines it is essentially equivalent to a Madaline, i.e., a single-layer network. This is caused by the linearity of the neuron transfer function in the Adaline implying that $y_i^p = \sigma_i^p$. This can be shown by introducing $w_{ij}^{p\,(n)}$ as the n-th order weight connecting y_j^{p-n-1} to y_i^p. We note that $w_{ij}^{p\,(0)} = w_{ij}^p$. A multi-layer network consisting of Adalines collapses into a single layer having weights $w_{11}^{N\,(N)}, \ldots, w_{m_N m_0}^{N\,(N)}$. We illustrate this by showing that y_i^p is directly expressible as a linear summation of $y_1^{p-2}, \ldots, y_{m_{p-2}}^{p-2}$ using first-order weights between layer $p-2$ and p.

$$
\begin{aligned}
y_i^p &= \sum_{j=1}^{m_{p-1}} w_{ij}^p y_j^{p-1} = \sum_{j=1}^{m_{p-1}} w_{ij}^p \sigma_j^{p-1} \\
&= \sum_{j=1}^{m_{p-1}} w_{ij}^p \sum_{k=1}^{m_{p-2}} w_{jk}^{p-1} y_k^{p-2} = \sum_{k=1}^{m_{p-2}} \underbrace{\left(\sum_{j=1}^{m_{p-1}} w_{ij}^p w_{jk}^{p-1} \right)}_{w_{ik}^{p\,(1)}} y_k^{p-2} \\
&= \sum_{k=1}^{m_{p-2}} w_{ik}^{p\,(1)} y_k^{p-2} \qquad (7)
\end{aligned}
$$

The multi-layer structure does not collapse into a single layer if a non-linear output function is used, for instance, the hard-limiting threshold function used in the Perceptron neuron processing model (cf. Equation 1). However, as we indicated in the previous section, this step function is not differentiable and, hence, it does not allow the adaptation of weights using a gradient descent. Therefore, the step function is substituted by a *sigmoid* function having a similar shape but with a continuous derivative (cf. Figure 6(a)). This results in a multi-layer network with neurons computing the sigmoidal function f of the weighted sum σ of their inputs (Rumelhart et al., 1986).

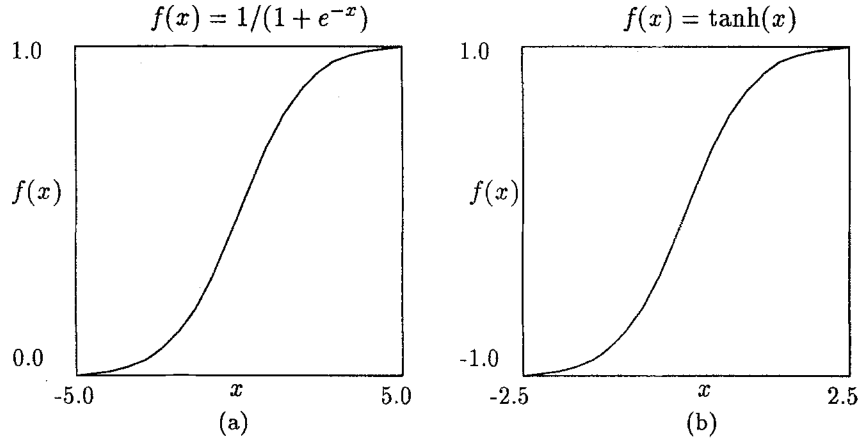

Fig. 6. Two typical output functions used in multi layer networks. (a) sigmoid function, (b) $\tanh(x)$ function.

The network output is obtained by propagating the input through the consecutive layers in Figure 4 until it reaches the output layer. Hence, this procedure is called *forward propagation*. If the inputs to the i-th neuron in layer p are $y_1^{p-1}, \ldots, y_{m_{p-1}}^{p-1}$ (i.e., the outputs from layer $p-1$) with corresponding weights $w_{i1}^p, \ldots, w_{im_{p-1}}^p$ then σ_i^p and the output y_i^p for this neuron are :

$$\sigma_i^p = \sum_{j=1}^{m_{p-1}} w_{ij}^p y_j^{p-1}$$
$$y_j^p = f(\sigma_i^p) = \frac{1}{1+e^{-\sigma_i^p}} \tag{8}$$

Sometimes other output functions are used, for instance, $\tanh(x)$ (cf. Figure 6(b)). The $\tanh(x)$ function is essentially identical to the sigmoid function and is particularly useful when network output should range between -1 and 1. We note that the sigmoid function depicted in Figure 6(a) ranges between 0 and 1.

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\frac{1}{1+e^{-2x}} - 1 = 2f(2x) - 1 \tag{9}$$

In the next section we describe how the gradient descent method underlying the delta rule used for the Madaline (cf. Section 3) may generalized such that it is suitable for configuring multi-layer neural networks with non-linear output functions.

4.2 Generalized Delta Rule

The *back-propagation* procedure (Rumelhart et al., 1986) is essentially a *gradient-descent* method which minimizes an error E by adapting weights (cf. Section 3). The error is measured as the sum of the squared errors of the actual response y_i^N and the desired (target) responses Y_i of the neurons in the output layer. For a single example E becomes:

$$E = \sum_{i=1}^{m_N} (y_i^N - Y_i)^2 \tag{10}$$

Although somewhat more complicated, this error function is essentially the same as the one presented in Equation 3 in Section 3. The error surface is defined as a function of the network parameters, i.e., the weights. Error E is minimized by a change (Δ) in the weights in the direction of the gradient descent, i.e., proportional to the negative gradient of E :

$$\Delta w_{ij}^p = -\eta \frac{\partial E}{\partial w_{ij}^p} \tag{11}$$

Constant η is called the *learning rate*, and is a positive real number. Increasing the learning rate on the one hand speeds up the adaptation process but on the other hand may cause the system to become unstable.

The derivative of E with respect to weights belonging to hidden layers is more difficult to determine because E is defined in terms of the error made by the *output* layer. However, it can be shown that the *error* in layer p can be expressed in terms of the errors occurring in the next layer $p+1$ and so on. The full derivation of the generalized delta rule is presented in Appendix A. This derivation introduces a *delta error* δ_i^p for all neurons in the network which is used to calculate the components of the error gradient.

$$\frac{\partial E}{\partial w_{ij}^p} = \delta_i^p y_j^{p-1} \tag{12}$$

The delta error δ_i^p is defined as the partial derivative of E with respect to the net input δ_i^p of neuron i in layer p.

$$\delta_i^p \overset{\text{def}}{=} \frac{\partial E}{\partial \sigma_i^p} \tag{13}$$

The partial derivative $\partial E / \partial \sigma_i^p$ is in fact a measure for the desired change in the output of the *specific* neuron j in order to minimize E. The delta error is spread through the network back from the last layer towards the first hidden layer directly following the input layer by a *back-propagation process.*

The derivation of the procedure for calculating δ_j^p in Equation 13 is described in Appendix A. It results in the *generalized delta rule* that can be used for calculating weights $w_{ij}^p(t)$ based on their value at previous iteration $t-1$ and the error:

$$w_{ij}^p(t) = w_{ij}^p(t-1) - \eta \delta_i^p y_j^{p-1} \tag{14}$$

$$\delta_i^p = \begin{cases} y_i^p(1-y_i^p) \sum_{j=1}^{m_{p+1}} w_{ji}^{p+1}\delta_j^{p+1} & 1 < p < N \\ y_i^N(1-y_i^N)(y_i^N - Y_i^N) & p = N \end{cases} \tag{15}$$

We note that the factor $y_i^p(1-y_i^p)$ in the calculation of the delta error corresponds to the derivative of the sigmoid function (cf. Equation 8). Namely, it can be shown that for the derivative of a sigmoid function $f(x)$ with respect to x, $f'(x) = f(x)(1-f(x))$ holds. In case the $\tanh(x)$ function (cf. Equation 9) is used, the factor $y_i^p(1-y_i^p)$ should be substitued by $1 - {y_i^p}^2$ since $\tanh'(x) = 1 - \tanh^2(x)$.

The Perceptron Learning procedure does not only adapt weights but also thresholds. It can be shown that the threshold adaptation rule in Equation 2 still applies when a threshold is entered in the sigmoidal function f (cf. Equation 8). Introducing a threshold avoids the situation where training is not very successful when $||\sigma|| \gg 0$. In that case f is almost horizontal and f' approaches 0. Hence, the weight change will also be close to zero. Moreover, a threshold may also avoid the emergence of local minima in the error surface (cf. Section 5). A *threshold* parameter θ is introduced for each neuron by using $g(\sigma, \theta) = f(\sigma - \theta)$ instead of just f in the processing model:

$$g(\sigma, \theta) = f(\sigma - \theta) = \frac{1}{1 + e^{-(\sigma - \theta)}} \tag{16}$$

As in Equation 11 the error E may be minimized by adapting θ. From Equations 8 and 16 it follows that the derivative of $g(\sigma, \theta)$ with respect to θ equals $-f'(\sigma - \theta)$. In Appendix A it is shown that the mathematical form of the Perceptron Learning rule for adapting the threshold (cf. Equation 2) still holds, viz.:

$$\Delta\theta = \eta\delta \tag{17}$$

In many implementations of the Back Propagation procedure, neurons have no explicit thresholds. Instead, a so-called *bias* neuron is added to the network. The bias neuron receives no input and constantly has output -1. Each neuron has a connection to the bias neuron. The adaptation (cf. Equation 12) of the corresponding connection weight reduces to the threshold adaptation in Equation 17. Hence, the weight to the bias neuron has the same functionality as the threshold.

5 Local minima

A gradient descent procedure searches for a minimal error on the error surface (cf. Figure 3). Once a minimum is reached there is no way out regardless of the fact that other, better, minima may exist. If a better minimum exists the current position is located in a *local minimum* of the error space. If, however, it is the lowest point among all then we speak of a *global minimum*. The possible occurrence of non-global minima is a well known problem that one has to be aware of using a gradient descent procedure (see also the contribution by Lenting and the contribution by Crama et al.). In order to understand this phenomenon, a multi-layer neural network (cf. Section 4) with three neurons and two weights is studied (McClelland and Rumelhart, 1988). The network contains one input, one hidden and one output neuron, hence it is called a 1:1:1 network (cf. Figure 7).

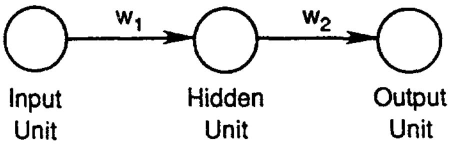

Fig. 7. A 1:1:1 network with one input, one hidden, one output neuron and two weights.

Here, the problem is to configure w_1 and w_2 such that the identity mapping is realized for binary input, i.e., the output should turn on if the input is turned on and it should turn off if the input is turned off. For the network in Figure 7 there exist two solutions to this problem. The first solution is straightforward. The hidden neuron simply propagates the unchanged on/off input signal and so does the output neuron. The other solution is that the hidden neuron transfers the opposite of the input neuron and the output neuron transfers the opposite of the hidden neuron. Obviously, taking two times the opposite of either *on* or *off* will result in the same signal. Both solutions correspond to global minima. In Table 3 weight configurations are presented that will finally converge and approach zero error when further adapted.

Table 3. Weight and Bias Configuration for the solutions of the identity mapping in a 1:1:1 network.

Solution	w_1	w_2	$bias_1$	$bias_2$
straight	-8	-8	$+4$	$+4$
not-not	$+8$	$+8$	-4	-4

The gradient descent procedure can find appropriate configurations for the identity mapping without getting trapped in a local minimum. However, if the biases are fixed at zero, a local minimum appears in the error surface. The error function can be calculated by summing the errors of the network for the two possible situations, i.e. $x = 0, y = 0$ and $x = 1, y = 1$. Using the process model of a multi-layer neural network (cf. Equation 8) we arrive at the following error function $E(w_1, w_2)$:

$$E(w_1, w_2) = (1 + e^{-w_2/2})^{-2} + (1 - 1/(1 + e^{-w_2/(1+e^{-w_1})}))^2 \qquad (18)$$

The error surface is plotted in Figure 8 for $w_1, w_2 \in [-10, 10]$ for two different viewpoints. In figure (a) the saddle point is very clear. The global minimum is at the foreground. In figure (b) the view is changed to emphasize the left side of the "saddle" which is actually a descent to a local minimum. Although this a very smooth descent it makes it impossible for a gradient descent procedure to get to the other side of the "saddle".

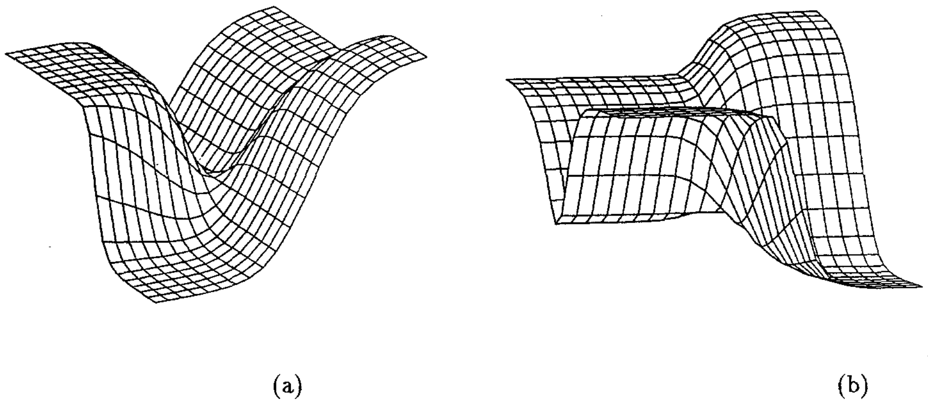

Fig. 8. The error surface of the identity mapping for a 1:1:1 network with biases fixed at zero. Figure (a) shows a saddle point with the global minimum on the foreground, (b) shows a different view indicating the existence of a local minimum at the left side of the "saddle".

The existence of local minima can very easily lead to a failure of the gradient descent search. If such a situation occurs one could try starting from a different initial weight setting. Fortunately, it seems that the error surface of a network with many weights has very few local minima. Apparently, in such networks it is always possible to slip out the local minimum by some other dimension. A more reliable method for escaping from local minima in a gradient search is called *simulated annealing* (Kirckpatrick et al., 1983). Normally, it is not possible to go uphill in a gradient descent. When applying simulated annealing every adaptation is performed with a certain probability. This introduces the possibility of going uphill, enabling an escape from local minima. Since it is more probable of getting out of a less deep minimum by chance the system is most likely to end in a global minimum instead of a local minimum. In simulated annealing this process converges by slowly "freezing" the system, i.e., by decreasing the probability of adaptation. A similar strategy is applied in the Boltzmann neural network (see the contribution by Spieksma).

6 Enhancements

The Back Propagation procedure converges very slowly, which is typical for many gradient descent procedures. Moreover, when the number of neurons increases linearly, the speed decreases more than linear. This is caused by the fact that the dimension of the gradient equals the number of weights in the network. In a multi-layer network the number of weights is roughly equal to the square of the number of neurons in the largest layer. On the other hand, however, we saw that if the dimension of the error space increases there seems to be a smaller chance of getting trapped in a local minimum. If we were able to speed up the learning process, Back Propagation would be very useful for configuring large networks as well. Increasing the learning rate does not always speed up the learning process. As a matter of fact, if the learning rate becomes too large, the system will certainly begin to oscillate and the learning process halts. In this section we will describe three alternative methods for speeding up the learning process. The first method uses a so-called *momentum* (Rumelhart et al., 1986), the second is called the *adaptive back-propagation algorithm* (Silva and Almeida, 1990), and the third is called *Super SAB, a self-adapting Back Propagation Algorithm* (Tollenaere, 1990).

6.1 Momentum

The weight adaptation described in Equation 15 is very sensitive to small disturbances. Suppose the direction of the gradient changes due to, for example, a bump in the error surface. In that case the back propagation procedure may just as well continue by going straight over the bump since it will vanish quickly. If, however, the gradient change is persistent, the adaptation will take notice of it. This strategy is accomplished by taking into account the previous adaptations in the learning process so that it gets a *momentum*. In practice this means that the weight adaptation calculated at step t (cf. Equation 15) is combined with the adaptation from step $t-1$ multiplied with a so-called *momentum parameter* α. The adaptation process then becomes :

$$\Delta w_{ij}^{p}(t) = -\eta \delta_i^{p+1} y_j^p + \alpha \Delta w_{ij}^{p}(t-1) \tag{19}$$

The momentum parameter α has to be in $[0, 1)$ otherwise the contribution of each Δw_{ij}^p grows infinitely. On the other hand, if α is too small the momentum becomes insignificant. One should therefore set the value of α close to one, e.g., 0.9. A basic problem with the momentum method is that it assumes the gradient slowly decreases when arriving close to the minimum. If, however, this is not the case the adaptation process will go through the minimum at high speed due to its momentum.

6.2 Adaptive Back-Propagation algorithm

When looking at an error surface, see, for example, Figure 8, we see that slopes may be gentle in one direction but steep in another. If the gradient descent travels by a small gradient, learning is slow. In such cases it would be a good idea to use a higher learning rate. On the other hand, if the gradient is steep the learning rate should be kept small. This strategy is accomplished by assigning to each weight w an individual learning rate η that is increased if the sign of its gradient component remains the same for some iterations, and is decreased otherwise. If $\Delta w_i^p(t-1)$ and $\Delta w_i^p(t)$ are the weight changes at time $t-1$ and t respectively and $\eta_i^p(t-1)$ is the corresponding learning rate at $t-1$, then the new learning rate may be calculated as follows:

$$\eta_i^p(t) = \begin{cases} \mu\eta_i^p(t-1) \text{ if } \Delta w_i^p(t)\Delta w_i^p(t-1) \geq 0 \\ d\eta_i^p(t-1) \text{ if } \Delta w_i^p(t)\Delta w_i^p(t-1) < 0, \end{cases} \tag{20}$$

Constants μ and d are an increase and a decrease factor respectively.

Silva and Almeida (1990) note that in a wide range of tests performed with this technique, they found that a value of μ somewhere between 1.1 and 1.3 was able to provide good results. For the parameter d, a value slightly below $1/\mu$ enables the adaptive process to give a small preference to learning-rate decrease, yielding a somewhat more stable convergence process. As one might expect, this technique may cause problems due to the fact that gradient components are changed independent from each other. This problem may be avoided by testing the total output error after adaptation has taken place. If there is an increase in error the new adaptation is rejected and a new set of learning rates is calculated using the gradient of the rejected adaptation. If this simple strategy does not work after a few trials, then it is always possible to simply reduce all the learning rate parameters by a fixed factor and repeat the process.

6.3 SuperSAB

SuperSAB (from Super Self-Adapting Back propagation) (Tollenaere, 1990) is a combination of the momentum method and adaptive back propagation. This algorithm is based on adaptive Back Propagation with a momentum. In each step the learning rate is increased exponentially using μ (see adaptive back propagation). When the sign of a gradient component changes the responsible adaptation is cancelled using the momentum. This is an important difference with the original adaptive back-propagation algorithm where learning is only slowed down but where the last weight adaption is not cancelled after a gradient change. Furthermore, the learning rate is decreased exponentially using d (see adaptive back propagation). Before Back Propagation is continued, the momentum should be set to zero to avoid making the same mistake again. Experiments with Super SAB indicate that in many cases the algorithm converges faster than gradient descent. In all cases the algorithm is less sensitive to parameter values than the original back propagation algorithm.

7 Simple recurrent network

The networks described in the previous sections are limited to realizing static mappings. Once a network is configured it only maps the input at time t on the output according to some learned mapping. Hence, the network is not capable of taking into account inputs that it processed earlier unless they were presented during the learning period.

One way to eliminate this restriction is to use a shift register that consists of N buffers, each capable of storing a single value. Each time a new input sample arrives the buffers are shifted, i.e., buffer N becomes $N-1$, $N-1$ becomes $N-2$ etcetera. The contents of buffer N are forgotten and the new sample is stored in buffer 1. A network with N buffers as input neurons is capable of processing temporal information restricted to the last N samples. First of all this solution is very awkward since it requires shift registers that are physically limited, meaning that only a limited number of samples can be retained. Secondly, this solution introduces a translation problem since a pattern can begin at N different positions.

Another way to eliminate this restriction without using buffers is to use outputs at time $t-1$ as input at time t (see also the contribution by Weijters and Hoppenbrouwers). These may either be outputs from neurons in the output layer but can just as well be taken from any other neuron in the network. Such a network is called a *recurrent network* and since its structure has remained the same it can still be trained using the Back Propagation procedure. One particular kind is called *Simple Recurrent Network* (SRN) and has been studied by Elman (1988). This network contains an input layer, a hidden layer and an output layer. The input layer is divided into *input neurons* that actually serve as the network input and so-called *context neurons* that are connected to the hidden neurons. For each hidden neuron there exists exactly one context neuron and after each iteration the output of a hidden neuron is copied to the output of its corresponding context neuron. The structure of this recurrent network is depicted in Figure 11.

8 Robot Arm

In this section the Back Propagation procedure is used to configure a multilayer network for controlling a *robot-arm system*. After the network is adapted, or trained, it has an internal *model* enabling it to control the system. The robot-arm setup is drawn in Figure 9. It depicts a robot arm that is bent at the shoulder over ϕ_1, and at the elbow over ϕ_2 degrees. Hence, the robot-arm is said to have two degrees of freedom. The problem is to find ϕ_1 and ϕ_2 such that the hand of the arm reaches at a point that coincides with the crossing point of the looking directions α_1 and α_2 of the left and right eye respectively.

In a real situation the examples, needed for training the system, may be obtained by taking measurements. In the case of Figure 9 this could, for example, be accomplished by using a mechanical model. After having placed the arm and eyes such that the eyes are looking at the hand, the corresponding angles can be measured. By repeating this procedure a collection of examples may be obtained. The advantage of this approach is that it is very straightforward to obtain examples for, e.g., a robot arm with five degrees of freedom. Deriving an analytical model for such an arm is still feasible although it is very difficult and certainly not cheap. We have chosen a robot arm with two degrees of freedom because it is relatively easy to derive an analytical solution. Looking at the complexity of the solution for this simple robot arm (cf. Table 4) gives a good idea of how complicated solutions for industrial robot arms may get.

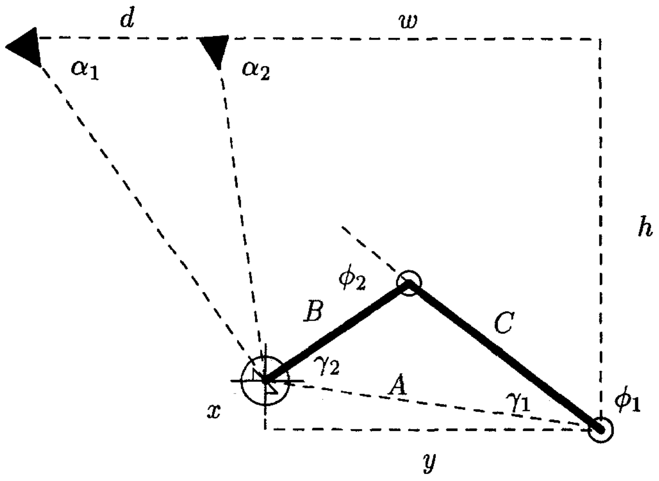

Fig. 9. Two eyes are rotated over α_1 and α_2 degrees respectively, looking at the hand of a robot arm with two freedoms ϕ_1 and ϕ_2 respectively.

The distance between hand and shoulder is denoted A. The length of the lower arm is B and of the upper arm is C. They determine the reach of the hand. We will use some addtitional variables enabling us to partition the transformation in 3 steps. In the first step the coordinates (x, y) of the hand are calculated relative to the shoulder. In the second step, angles γ_1 and γ_2 *inside* the "arm-triangle" are calculated. Finally, in the third step, the shoulder and elbow angles ϕ_1 and ϕ_2 are calculated.

Table 4. Analytical form of transformation for the robot arm.

I	$x = w - d \tan \alpha_1 / (\tan \alpha_2 - \tan \alpha_1)$ $y = h - d \tan \alpha_1 \tan \alpha_2 / (\tan \alpha_2 - \tan \alpha_1)$
II	$A^2 = x^2 + y^2$ $\cos \gamma_1 = (A^2 + C^2 - B^2)/2CA$ $\cos \gamma_2 = (A^2 - C^2 + B^2)/2BA$
III	$\phi_1 = 180^\circ - \arctan y/x - \gamma_1$ $\phi_2 = \gamma_1 + \gamma_2$

A multi-layer neural network can perform this mapping with reasonable accuracy after having learned a set of examples. In this case one may think of the network as an automatic interpolator. The development of this network may be divided in four phases that may be considered as a simple methodology for designing a multi-layer neural network that controls a robot arm.

8.1 Representation

A neural network may be thought of as a computer that programs itself according to a set of examples. This is not as good as it sounds. A neural network will only be capable of solving a problem if it gets all the information that is required. This means that, most of all, a neural network engineer must analyze the problem and determine what information is relevant and how it should be fed into the network. Moreover, he or she should also decide what kind of information is delivered by the network. This is a *representation* problem. Essentially, the representation problem in neural networks is caused by the large variety of possible representations. Choosing a correct representation does not only depend on the problem domain, it also depends on the physical capabilities of the neural network. For example, a multi-layer neural network will never be able to reach 1 as output owing to the sigmoidal function. If binary examples should be learned, it is wise to take, e.g., 0.1 instead of 0 and 0.9 instead of 1. In the robot-arm system there are two problems. The first problem is that angles range between 0 and 360 degrees. This can simply be solved by scaling the angles to the range $[0, 1]$, dividing the angles by 360. The second, more serious problem, is that angles are periodic, i.e. 0 equals 360, or, after scaling, 0 is 1. This makes it very difficult, if not impossible, for a network to learn a proper model, because 0 and 1 have opposite meanings by their very nature. There are two ways to deal with this problem:

1. The first solution is that by making sure that the hand can only reach in front of the eyes, ($A \leq h$, cf. Figure 9), we ensure that $\alpha_1, \alpha_2 \in [0°, 180°]$. After scaling we obtain a representation in which 0 and 1 actually, and literally, have an opposite interpretation. However, when using this solution, it would mean that the angles of the arm are also restricted to this range unless other output features are added that, for example, represent the sign of ϕ_1 and ϕ_2.
2. The second solution is to represent angles as a sine, cosine pair. This means, however, that instead of two there are four inputs and four outputs. Due to this solution the scaling problem has slightly changed since the values of sine and cosine range between -1 and 1. This can be solved by adding 1 and dividing the result by 2. Using this representation 0 and 1 have opposite interpretations. We note that the linear scaling to $[0, 1]$ is not necessary if the $\tanh(x)$ function is used instead of the sigmoid function.

We have chosen for the second solution since it is straightforward and the sine and cosine of the viewing directions can be measured directly when obtaining examples from a mechanical model. Now that we have established the representation we can proceed by generating a collection of examples.

8.2 Examples

In order to train the network with the Back Propagation procedure (cf. Section 4) it is necessary to have a collection of examples. A single example consists of an array with input values and an array with output, or target, values.

When training the network, an example is selected and is fed into the network. The output is compared to the target and subsequently the error for this particular example made by the network is calculated. With this error the network can be adapted using the generalized delta rule (cf. Equation 15). The shape of the examples is determined by the representation. Hence, in this case, examples will consist of eight floating-point numbers, i.e., ($\sin\alpha_1$, $\cos\alpha_1$, $\sin\alpha_2$, $\cos\alpha_2$, $\sin\phi_1$, $\cos\phi_1$, $\sin\phi_2$, $\cos\phi_2$). Using the analytical solution described in Table 4 a set of examples can be generated. The robot arm should perform equally well for all points. Hence, it is necessary to select at random pairs (α_1, α_2) in the area that can be reached so that the examples have a uniform distribution. If, for example, they are not uniformly distributed but all situated at the left side of the shoulder, the arm is not likely to learn positions at the right side of the shoulder.

In addition to the example set it is useful to have a test set that is constructed in the same way but which contains different examples. Calculating the sum of the squared errors of the examples in the test set provides us with a measure of the overall performance of the network. If this error is low then there are good reasons for assuming that the network has created a generalized model since the samples in the test set were not really learned.

8.3 Configuration

Before training can begin we have to configure the network. The number of input and output neurons is already determined by the representation. Determining the number of hidden layers the number of hidden neurons in the network is normally a difficult task that can only be accomplished by trial and error. However, in this case, we are dealing with a continuous mapping (cf. Table 4) and only one hidden layer is needed. Namely, according to Lippmann (1987) a three layer perceptron with $N(2N+1)$ nodes using continuously increasing non-linearities can compute any continuous function of N variables. Unfortunately, the theorem does not indicate how weights or non-linearities in the network should be selected or how sensitive the output function is to variations in the weights and internal functions.

Based on Lippmann's statement we assume that one hidden layer will suffice. However, it is not exactly clear how many neurons it should contain since the network has four outputs. Next, we will show that by using the error of the test set it is possible to indicate when the number of hidden neurons is not sufficient.

8.4 Learning

The rules presented in this section are based on experience. Hence, their validity can not be proven but in general they may be used for developing a useful learning strategy.

During the learning phase it is important to monitor the error of the example set and of the test set. Learning should always proceed if there is still a considerable reduction of the error. Only when the changes get very small, i.e., learning becomes tediously slow, one should start wondering if the learning goal has been achieved or if a problem has come up. We will discuss a number of situations that may occur.

1. Normally, the errors on both the example and the test set should decrease continuously with small disturbances. If, however, these disturbances are large and very frequent this probably means that the system is unstable. In that case the learning-rate parameter should be decreased.
2. If the error of the example set has become, or is almost zero, then the learning phase should be halted because no more can be learned from the current examples. In this case there are two possibilities: (1) the error on the test set is also zero or almost zero, and (2) the error on the test set is still considerable. In the first situation the network has successfully completed the learning task. In the second situation there is a good indication that there are not enough examples or that they are not representative for the problem domain. It is advisable to review the examples and try again.
3. If shortly after the beginning the error on the example set is decreasing very slowly but is still considerable there may be a problem. First of all it may be that the learning rate is too small. A typical learning rate is 0.3. By increasing the learning rate the error should decrease more rapidly. However, if this leads to unstable behaviour, it may be that the examples are inconsistent or that an inefficient representation was chosen. For instance, mapping 0° on 0 and 360° on 1 is inefficient because 0 and 1 are opposite neuron activity levels while 0° and 360° represent the same angle. Inconsistencies in the example set may be found by examining if there are contradictions. In that case an alternative representation should be considered. Another explanation for not being able to learn the examples may be that the network has not enough hidden neurons. Hence, if the representation seems correct and the error is really not going to become zero, then it is advisable to try again with more hidden neurons.
4. If the errors are non zero and do not change, this could mean the learning process is trapped in a local minimum (cf. Section 3) or it could indicate that there is no solution. One way to deal with this problem is to use another initial set of weights and start all over again hoping that a local minimum will be avoided.
5. It may be possible that both the example error and test error are decreasing smoothly but that suddenly the learning rate drops drastically. After a while the example error decreases again but the test error only increases. A possible explanation is that until the moment the learning rate dropped the network was perfectly well capable of creating a model. Then, suddenly, the network is not able to gain more accuracy and learning stops. Apparently the network tries to achieve the desired accuracy by learning a perfect mapping without

generalizing. Hence, the error of the test set increases. By adding more hidden neurons this problem may be avoided.

8.5 Simulation Results

The following simulation results were obtained with a Back Propagation training procedure using only the momentum ehancement. Using the analytical solutions stated in Table 4 a learning set and a test set were constructed. Both consist of 100 randomly generated arm-eye orientations. The mean error on the examples of learning set and the test set during 1,000 training cycles are shown in Figure 10 (a) to (f) below.

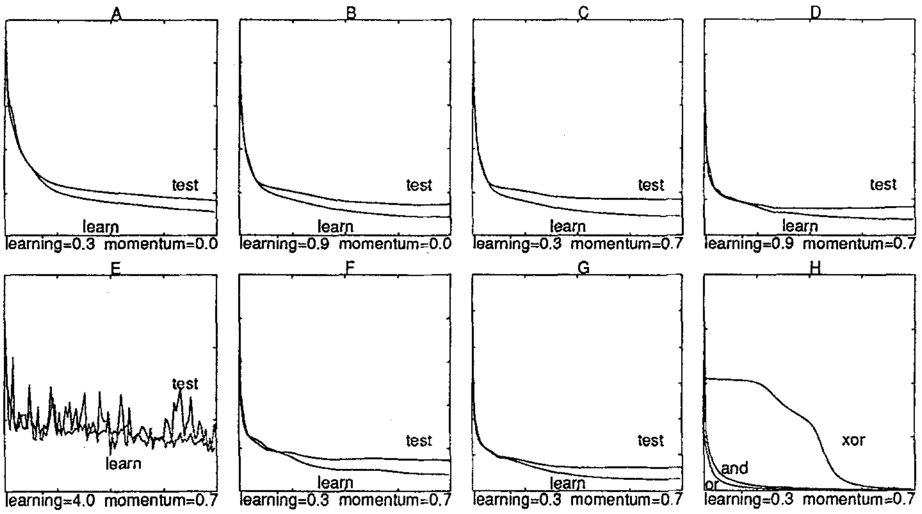

Fig. 10. Simulation results obtained by exucting 1,000 cycles over a randomly generated learning and test set. Figures (a)-(g) to the robot-arm application: (a), (b), (c) and (d) correspond to a 4:4:4 network, (e) shows instable behavior because α is too large; (f) and (g) correspond to a 4:8:4 and a 4:16:4 network respectively. Figure (h) shows the curves for 2:1 network trained with 'or' and 'and' problem, for a 2:2:1 network for the 'xor' problem.

Figures (a) to (e) were obtained from a 4:4:4 network. First of all it should be noticed that the error on the test set reduces quickly meaning that apparently there is a strong correlation between the examples in the learning set and those in the test set. Figures (a) and (b) were obtained without a momentum. Compared to (c) and (d), where the momentum is 0.9, the curves in (a) and (b) are initially less steep. The best results after 1,000 cycles were obtained in case (d) due to the higher learning rate. However, in figure (e) the learning rate is set to 4.0 which is too large. The result is that the curves start oscillating and that the learning rate is very slow or sometimes even negative. An adaptive Back Propagation algorithm like SuperSAB would detect the oscillations and immediately reduce the learning rate parameter. However, when the curve is smooth (e.g. figure (a)) SuperSAB increases the learning rate (cf. figure (b)). Figure (f) shows learning results for a 4:8:4 network and (g) for a 4:16:4 network. Although the learning rate parameter is set to 0.3 learning is even more quickly then in (d). It must be realized, however, that training a 4:16:4 network requires much more effort then training a 4:4:4 network, i.e., the true learning rate in (e) and (f) may actually be slower. It should be clear from these simulation results that in order to achieve a high learning rate it is important to tune the learning parameters. When looking at the performance of the robot arm when it is controlled by the network it is clear that the network can only interpolate from the examples it has learned.

In Figure 10(h) the simulation results for the 'or','and' and 'xor' problems are shown. The 'or' and 'and' problem were trained on a 2:1 network and converged very quickly. The 'xor' problem was trained on a 2:2:1 network and required considerable longer time to converge compared to the 'or' and 'and'. Very typical for the 'xor' problem is that it seems initially as if the network has got stuck in a local minimum but that after a few hundred cycles the error reduces quickly.

9 Conclusions

The most important conclusion to this paper is that multi-layer neural networks in combination with Back Propagation have a very wide area of application. Adaptive multi-layer neural networks form a useful technique for categorizing, recognizing and modeling data distributions. In such applications this technique must be considered as a serious competitor of classical (statistical) methods. One of the major advantages that makes this technique so desirable is that no *a priori* assumptions have to be made concerning the nature of the input distribution. However, in this paper it is also pointed out that some serious drawbacks exist. First of all, Back Propagation is a gradient descent in network weight space. Consequently, the search for solutions in this space may be hindered by local minima. Moreover, a good representation of the input and output of the network is essential. This usually involves a thorough analysis of the problem domain that may be just as difficult as designing an algorithmic solution. Furthermore, examples needed for training the network may not always be available. Another issue that has not yet been mentioned is that Back Propagation is rather difficult to realize in hardware compared to other network paradigms that rely on local adaptation rules. This means that when using Back Propagation, learning has to be done in advance and the resulting weights can be stored in a physical target neural network machine. For Back Propagation this is the only way to benefit from the virtue of neural networks, i.e., their ability to process large numbers of input in parallel.

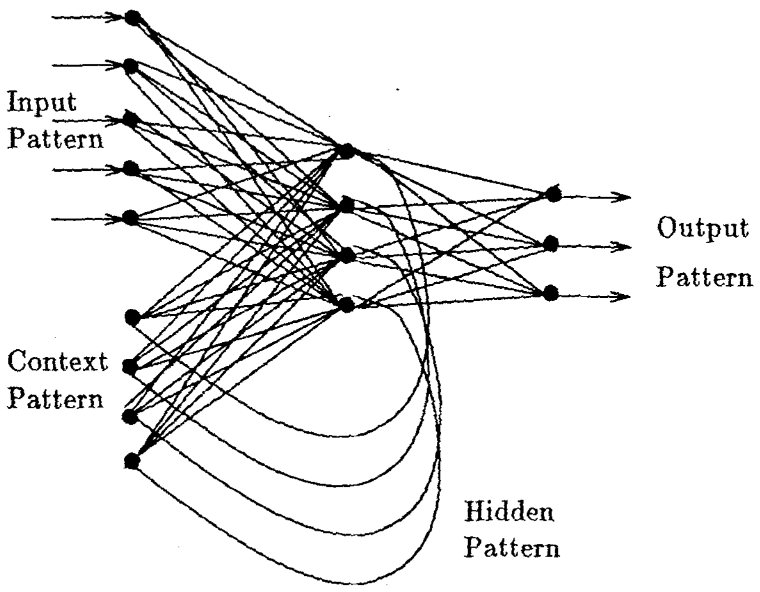

Fig. 11. A Simple Recurrent Network (Elman, 1988).

Appendix A: Derivation of the Generalized Delta Rule

In this Appendix a vectorized version of the generalized delta rule is derived. A vector notation is used because it enables us to exploit the regular structure of the network which will finally result in simpler rules. Moreover, Henseler and Braspenning (1990) use this result to prove that the generalized delta rule can also adapt multi-layered neural networks with complex-valued weights. Consequently, the net inputs to and the outputs from the neurons in layer p are denoted by vectors σ^p and y^p respectively:

$$\sigma^p = \begin{pmatrix} \sigma_1^p \\ \vdots \\ \sigma_{m_p}^p \end{pmatrix}, y^p = \begin{pmatrix} y_1^p \\ \vdots \\ y_{m_p}^p \end{pmatrix} \tag{21}$$

The lower indices in Equation 20 run over the neurons in larger p. Appropriately, the weights between layers $p-1$ and p in a multi-layer network can be written in a *weight matrix* W^p:

$$W^p = \begin{pmatrix} w_{11}^p & \cdots & w_{1m_{p-1}}^p \\ \vdots & \ddots & \vdots \\ w_{m_p1}^p & \cdots & w_{m_pm_{p-1}}^p \end{pmatrix} \tag{22}$$

Let vector $y^0(t)$ be the input in the network at time t. Generally we shall leave out the (t) index. The i-th component of y^0 denotes the activity of input neuron i. With y^0 the input in the network, the output y^N may be obtained by iteratively calculating $y^1, \ldots, y^N$, i.e. the output of layers $1, \ldots, N$ respectively. This process is called *forward propagation* of the network input. Equation 8 may be rewritten as follows:

$$\sigma^p = W^p y^{p-1} \tag{23}$$

$$y^p = f(\sigma^p) \tag{24}$$

The generalized-delta rule is based on a *gradient-descent* method which minimizes a total error E by adapting weights in the opposite direction of the gradient of the error surface in weight space. The error is measured as the sum of the squared errors of the actual responses $y_1^N, \ldots, y_{m_N}^N$ and the desired (target) responses $Y_1, \ldots, Y_{m_N}$ of the neurons in the output layer. In vector notation, the error function is equal to the squared length of the error vector, $Y - y^N$.

$$E = \sum_{i=1}^{m_N} (Y_i - y_i^N)^2 = ||Y - y^N||^2 \tag{25}$$

Since the network output y^N is determined by the connection weights, the error E is a function of the connection weights and will therefore be represented by $E(W^1, \ldots, W^N)$ which we will denote $E(W)$ for short. The objective is to find a network weight configuration $W^1, \ldots, W^N$ such that the error $E(w)$ is minimal.

A very simple method that may be used to approximate this minimum is to adapt $W^1, \ldots, W^N$ in the direction of the negative gradient. This method is called *gradient descent.* The error gradient contains components for each weight w_{ij}^p in the network. In a gradient descent, each weight w_{ij}^p is adapted proportionally (α) to the negative partial derivative of $E(w)$ with respect to w_{ij}^p:

$$\Delta w_{ij}^p \propto -\frac{\partial E(w)}{\partial w_{ij}^p} \tag{26}$$

We note, however, that gradient descent does not guarantee a global minimum to be found. It is only guaranteed that a local minimum is found. The generalized delta rule may be derived by rewriting Equation 26. Firstly, the chain rule is used to rewrite the partial derivative in the right-hand side:

$$\frac{\partial E(w)}{\partial w_{ij}^p} = \underbrace{\frac{\partial E(w)}{\partial \sigma_i^p}}_{\delta_i^p} \frac{\partial \sigma_i^p}{\partial w_{ij}^p} = \delta_i^p \underbrace{\frac{\partial}{\partial w_{ij}^p} \sum_{k=1}^{m_{p-1}} w_{ik}^p y_k^{p-1}}_{=0 \; k \neq j} = \delta_i^p y_j^{p-1} \tag{27}$$

The factor δ_i^p introduced in Equation 27 is called the *delta error* of the i-th neuron in layer p. The change for an entire weight matrix W^p is obtained by removing the indices i and j from Equation 27. This means that the weight change equals the product of δ in layer p and the output y^{T} of the previous layer $p-1$. The superscript $^{\mathrm{T}}$ denotes the transpose of a vector, i.e., y^{T} is a row vector and the product of δ and y^{T} is a matrix.

$$\Delta W^p = -\eta \delta^p y^{p-1\mathrm{T}} \tag{28}$$

Constant η is called the *learning rate*, and is a positive real number. Increasing the learning rate on the one hand speeds up the adaptation process but on the other hand may introduce oscillations in the learning process meaning that the descent along the error surface is not effective. One method that is often used to speed up the adaptation process without introducing oscillations is to modify Equation 28 by including a second-order term which is called a *momentum* (Rumelhart et al., 1986). The momentum represents a fraction of the previous weight adaptation. Let $\Delta W^p(n)$ denote the weight adaptation at the n-th iteration, then Equation 28 is modified as follows:

$$\Delta W^p(n+1) = -\eta \delta^p y^{p-1\mathrm{T}} + \alpha \Delta W^p(n) \tag{29}$$

The momentum coefficient $\alpha \in [0, 1)$ is a constant which determines the effect of past weight changes on current direction of the gradient descent. The momentum term filters high-frequency oscillations, i.e., it suppresses oscillations allowing the learning rate to be larger.

The derivative of the error $E(w)$ with respect to the net input σ_i^p in a neuron may be calculated by applying the chain rule again. Namely, we calculate the derivative of $E(w)$ with respect to y_j^p and multiply this with the derivative of y_j^p with respect to the net input σ_i^p of neuron j in layer p:

$$\delta_i^p = \frac{\partial E(w)}{\partial \sigma_i^p} = \sum_{j=1}^{m_p} \frac{\partial E(w)}{\partial y_j^p} \frac{\partial y_j^p}{\partial \sigma_i^p} \tag{30}$$

Allthough the partial derivative of y_j^p to σ_i^p is zero if $i \neq j$ the summation over $j = 1, \ldots, m_p$ in Equation 30 is entered to formulate δ_i^p as the inner product of two vectors:

$$\delta_i^p = \left(\frac{\partial E(w)}{\partial y_1^p}, \ldots, \frac{\partial E(w)}{\partial y_{m_p}^p} \right) \begin{pmatrix} \partial y_1^p / \partial \sigma_i^p \\ \vdots \\ \partial y_{m_p}^p / \partial \sigma_i^p \end{pmatrix} \tag{31}$$

This equation can be extended to a matrix multiplication resulting in $\delta^{p\mathrm{T}} = (\delta_1^p, \ldots, \delta_{m_p}^p)$:

$$\delta^{p\mathrm{T}} = \left(\frac{\partial E(w)}{\partial y_1^p}, \ldots, \frac{\partial E(w)}{\partial y_{m_p}^p} \right) \underbrace{\begin{pmatrix} \partial_1^p / \partial \sigma_1^p & \ldots & \partial_1^p / \partial \sigma_{m_p}^p \\ \vdots & & \vdots \\ \partial_{m_p}^p / \partial \sigma_1^p & \ldots & \partial_{m_p}^p / \partial \sigma_{m_p}^p \end{pmatrix}}_{\Psi^p} \tag{32}$$

Matrix Ψ^p is called the layer differential matrix and the delta error δ^p in layer p is calculated as follows:

$$\delta^p = \Psi^{p\mathrm{T}} \frac{\partial E(w)}{\partial y^p} \tag{33}$$

Neurons in a multi-layer network that are located in the same layer are not interconnected; hence, matrix Ψ^p is diagonal since the i-th component of y^p depends on σ_i^p only, i.e., $\partial y_j^p / \partial \sigma_i^p = 0$ if $i \neq j$. It can be shown from the definition of f in Equation 8 that if $y = f(\sigma)$ then $y' = f'(\sigma) = y(1-y)$. Hence, Ψ^p can be presented as :

$$\Psi^p = \begin{pmatrix} y_1^p(1-y_1^p) & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & y_{m_p}^p(1-y_{m_p}^p) \end{pmatrix} \tag{34}$$

The derivative of $E(w)$ with respect to y_j^p in Equation 30 can only be calculated directly for $p = N$, i.e., for the output layer. This was expected since the error is measured in the output layer for which the target values $\mathbf{Y}$ are specified by the example. According to the definition of $E(w)$ in Equation 25, the derivative for the output layer is :

$$\frac{\partial E(w)}{\partial y^N} = -2(Y - y^N) = 2(y^N - Y) \tag{35}$$

The factor 2 will be left out by entering it in the learning rate η (cf. Equation 29). The delta error in layer N can be calculated using Equation 33. The error δ^p in hidden layer p is calculated in terms of the error made by the subsequent layer $p+1$ (up to N) as follows :

$$\frac{\partial E(w)}{\partial y_i^p} = \sum_{j=1}^{m_{p+1}} \underbrace{\frac{\partial E(w)}{\partial \sigma_j^{p+1}}}_{\delta_j^{p+1}} \frac{\partial \sigma_j^{p+1}}{\partial y_i^p} = \sum_{j=1}^{m_{p+1}} \delta_j^{p+1} \underbrace{\frac{\partial}{\partial y_i^p} \sum_{l=1}^{m_p} w_{jl}^{p+1} y_l^p}_{=0 \text{ if } l \neq i}$$

$$= \sum_{j=1}^{m_{p+1}} \delta_j^{p+1} w_{ji}^{p+1} \tag{36}$$

This result enables the formulation of the generalized delta rule for δ_i^p in its usual form by substituting Equation 36 into Equation 30 and denoting $\partial y_i^p / \partial \sigma_i^p = f'(\sigma_i^p)$:

$$\delta_i^p = f'(\sigma_i^p) \sum_{j=1}^{m_{p+1}} \delta_j^{p+1} w_{ji}^{p+1} \tag{37}$$

Apparently, the error propagation according to Equation (37) looks like forward propagation (cf. Equation 24) except that it is in the opposite direction, hence called *back propagation.* We will rewrite Equation 36 as a matrix multiplication resulting in $(\partial E(w)/\partial y^p)^{\mathrm{T}} = (\partial E(w)/\partial y_1^p, \ldots, \partial E(w)/\partial y_{m_p}^p)$:

$$\left(\frac{\partial E(w)}{\partial y^p}\right)^{\mathrm{T}} = (\delta_1^{p+1}, \ldots, \delta_{m_{p+1}}^{p+1}) \begin{pmatrix} w_{11}^{p+1} & \cdots & w_{1m_p}^{p+1} \\ \vdots & & \vdots \\ w_{m_{p+1}1}^{p+1} & \cdots & w_{m_{p+1}m_p}^{p+1} \end{pmatrix} \tag{38}$$

which reduces further to,

$$\frac{\partial E(w)}{\partial y^p} = W^{p+1^{\mathrm{T}}} \delta^{p+1} \tag{39}$$

Combining the results found in Equations 33, 35 and 39 the following equations can be formulated for calculating the delta error vector δ^p :

$$\delta^p = \begin{cases} \Psi^{p^{\mathrm{T}}} W^{p+1^{\mathrm{T}}} \delta^{p+1} & 1 < p < N \\ \Psi^{N^{\mathrm{T}}} (y^N - Y) & p = N \end{cases} \tag{40}$$

Equation 40 is also referred to as the *generalized delta rule* and is used when calculating the weight change ΔW defined in Equation 29. This rule is a generalized version of the *delta rule* (cf. Equation 2) used for adapting weights in layered networks without hidden layers, i.e., Perceptrons (Rosenblatt 1958).

It can be shown that the generalized delta rule also applies to threshold adaptation when a threshold parameter is added to the sigmoidal function (cf. Equation 16), i.e., Equation 24 becomes $y^p = f(\sigma^p - \theta^p)$. If the j-th neuron in layer p has threshold θ_j^p then the threshold change (Δ) to minimize the error $E(w)$ is given by:

$$\Delta\theta_i^p = -\eta\frac{\partial E(w)}{\partial\theta_i^p} = -\eta\frac{\partial E(w)}{\partial y_i^p}\frac{\partial y_i^p}{\partial\theta_i^p} = -\eta\frac{\partial E(w)}{\partial y_i^p}\frac{-\partial y_i^p}{\partial\sigma_i^p} = \eta\delta_i^p \tag{41}$$

We note that the final transition is accomplished by substituting Equation 30 taking into account that $\partial y_j^p/\partial\sigma_i^p = 0$ if $i \neq j$. Using a vector notation and adding a momentum (cf. Equation 29) the following threshold adaptation rule is used:

$$\Delta\theta^p(n+1) = \eta\delta^p + \alpha\Delta\theta^p(n) \tag{42}$$

Appendix B: Back Propagation algorithm

```
MAINLOOP

  randomize weights
  repeat

    total error = 0
    for all examples $\{X, Y\}$

      forward propagation($X$)
      back propagation($Y$, error)
      total error = total error + error
    endfor

  until total error $< \epsilon$

END MAINLOOP
```

```
FORWARD PROPAGATION

INPUT: $x_1, \ldots, x_{m_0}$
OUTPUT: -

  $y^0 = x$
  for $p = 1$ to $N$

    for $i = 1$ to $m_p$

      $\sigma = -\theta_i^p$
      for $j = 1$ to $m_{p-1}$

        $\sigma = \sigma + w_{ij}^p y_j^{p-1}$
      endfor

      $y_i^p = f(\sigma)$
    endfor
  endfor

END FORWARD PROPAGATION
```

```
BACK PROPAGATION

INPUT : $Y_1, \ldots, Y_{m_N}$
OUTPUT: error

  error = 0
  for $i = 1$ to $m_N$

    $\delta_i^N = y_i^N(1 - y_i^N)(y_i^N - Y_i)$
    error = error + $(y_i^N - Y_i)^2$
  endfor

  for $p = N - 1$ downto 1

    for $i = 1$ to $m_p$

      $\sigma = 0$
      for $j = 1$ to $m_{p+1}$

        $\sigma = \sigma + w_{ji}^{p+1}\delta_j^{p+1}$
      endfor

      $\delta_i^p = y_i^p(1 - y_i^p)\sigma$
    endfor
  endfor

  for $p = N$ downto 1

    for $i = 1$ to $m_p$

      $\hat{\theta}_i^p = \eta\delta_i^p + \alpha\hat{\theta}_i^p$
      $\theta_i^p = \theta_i^p + \hat{\theta}_i^p$
    endfor

    for $i = 1$ to $m_{p-1}$

      for $j = 1$ to $m_p$

        $\hat{w}_{ji}^p = -\eta\delta_j^p y_i^{p-1} + \alpha\hat{w}_{ji}^p$
        $w_{ji}^p = w_{ji}^p + \hat{w}_{ji}^p$
      endfor
    endfor
  endfor

END BACK PROPAGATION
```

References

J.L. McClelland and D.E. Rumelhart (1988) Training hidden units: the Generalized Delta Rule. Chapter 5 in *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises.* MIT Press, Cambridge, MA.

J.L. Elman (1988) Finding structure in time. CRL Technical Report 8801. Center for research in Language, University of California, San Diego.

J. Henseler and P.J. Braspenning (1990) Training complex multi-layer neural networks, *Proceedings of the Latvian Signal Processing International Conference*, Vol. 2, Riga, 301–305.

S. Kirckpatrick, C.D. Gelatt and V. Torre (1983) Optimization by simulated annealing. *Science* 220, 671–680.

Y. Le Cun (1985) A Learning Procedure for Assymetric Threshold Network. *Proceedings of Cognitiva '85.* (In French), Paris. 599–604.

R.P. Lippmann (1987) An introduction to computing with neural nets. *IEEE ASSP Magazine* 3 (4), 4–22.

M.L. Minsky en S.A. Papert (1969, 1988) *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, Cambridge, MA.

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

F. Rosenblatt (1958) The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65, 386–408.

F. Rosenblatt (1962) *Principles of Neurodynamics.* Spartan, New York.

D.E. Rumelhart, G.E. Hinton and R.J. Williams (1986) Learning internal representations by error propagation. Chapter 8 in *Parallel Distributed Processing : Foundations.* Vol. 1, MIT Press, Cambridge, MA, 318–362.

F.M. Silva and L.B. Almeida (1990) Acceleration techniques for the Backpropagation algorithm. *Lecture Notes in Computer Science: Neural Networks* 412 (Eds. L.B. Almeida and C.J. Wellekens), 110–119.

T. Tollenaere (1990) SuperSAB, fast adaptive Back Propagation with good scaling properties. *Neural Networks* 3 (5), 561–573, Pergamon-Press.

P. Werbos (1974) Beyond Regressions New Tools for Prediction and Analysis in the Behavioral Sciences. M. Sc. thesis, *Applied Mathematics*, Harvard University, Boston, MA.

B. Widrow and M.E. Hoff (1960) Adaptive switching circuits. *Record of the 1960 IRE WESCON Convention*, New York, IRE, 96–104.