Backpropagation Algorithms : Their Characteristics and Efficiency

Marcus Pfister Raúl Rojas

B 3 - 93

May 3, 1993

Abstract

Backpropagation is the most popular learning rule for multilayer neural networks. The algorithm is easy to understand and implement, but converges rather slowly when dealing with problems above a certain complexity threshold. In recent years much effort has been spent trying to develop more efficient variations of backpropagation. This has lead to a combinatorial explosion of different learning algorithms of which no detailed overview exists. This paper provides a classification of the main variations of backpropagation and gives some results on their relative efficiency. We addressed the difficult problem of comparing the algorithms by selecting a set of benchmarks, which we consider representative of difficult learning problems. The performance of each algorithm was tested using these benchmarks, and we report our results.

Freie Universität Berlin
Fachbereich Mathematik – Institut für Informatik
Takustr. 9
1000 Berlin 30

e-mail: pfister@inf.fu-berlin.de, rojas@inf.fu-berlin.de

Backpropagation Algorithms : Their Characteristics and Efficiency

Marcus Pfister Raúl Rojas

May 3, 1993

Abstract

Backpropagation is the most popular learning rule for multilayer neural networks. The algorithm is easy to understand and implement, but converges rather slowly when dealing with problems above a certain complexity threshold. In recent years much effort has been spent trying to develop more efficient variations of backpropagation. This has lead to a combinatorial explosion of different learning algorithms of which no detailed overview exists. This paper provides a classification of the main variations of backpropagation and gives some results on their relative efficiency. We addressed the difficult problem of comparing the algorithms by selecting a set of benchmarks, which we consider representative of difficult learning problems. The performance of each algorithm was tested using these benchmarks, and we report our results.

1 Introduction

Backpropagation, or backpropagation-like algorithms, were developed by several researchers working independently between 1970 and 1985. Backpropagation was originally used by A. E. Bryson and Yu-Chi Ho in 1969 [BH69] and was independently rediscovered by P. J. Werbos in 1974 [Wer74] and still later by D. P. Parker in 1985 [Par85]. But it was mainly the work of the PDP group around Rumelhart, Hinton and McClellan [RHW86], that made backpropagation, which they called *the generalized delta rule*, really popular. The published work of the PDP group provided a cristallization point for a new appreciation of the algorithm. For more details about the colorful history of backpropagation see [HN89, HN91, Wer88].

Before backpropagation was introduced, there had been attempts to train rather simple neural networks with only one or two layers of perceptrons. Minsky and Papert, who provided the first careful analysis of the mapping properties of these networks, concluded, that there were a large number of problems which could not be solved by a single layer of perceptrons [MP88]. They also showed, that networks with one ore more additional hidden layers (Figure 1) could be able to solve those problems.

The structure of backpropagation networks ist that of a directed graph. The nodes of the graph are computing units capable of evaluating certain primitive functions. Information flows between the nodes through the weighted edges of the graph. The information transported by each edge is multiplied with its associated weight. All incoming information to a node is added and the node's primitive function is evaluated. An input vector (also called a *pattern*) fed into some nodes of the network starts a chain of evaluations and the final result at some selected nodes

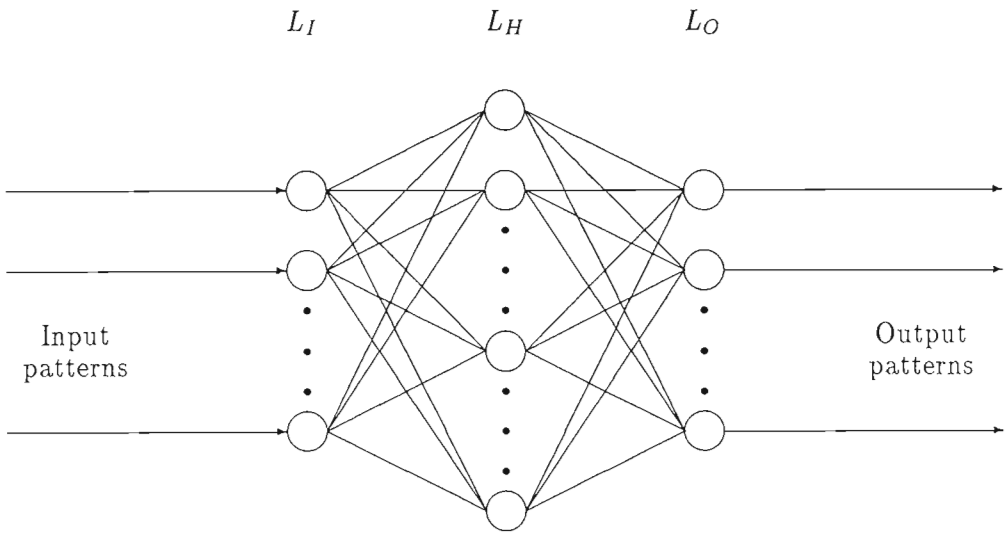


Figure 1: Network with one input-, one output- and one hidden layer (L_I, L_O and L_H).

(the output units) is recorded. The topology of the network and the primitive functions at the nodes are usually fixed. The only variable parameters in this computing system are the weights of the network.

The objective of building a backpropagation network is to compute a certain function of the input pattern which yields an output pattern. Different combinations of weights implement different functions. The function to be computed is known only from a 'training set' of input-output pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$. Each x_i is an n-dimensional real vector and each y_j an m-dimensional one. We are faced with a typical optimization problem, namely to find those values of the network's weights which yield the best approximation to the unknown function as measured by the results on the training set. We thus can state some typical approximation problems of real functions in the framework of neural networks. If we want to approximate a function $f : \mathbb{R} \to \mathbb{R}$ with a polynomial of degree n, we need to determine the set of constants $a_0, a_1, \ldots, a_n$ such that $f(x) = a_0 + a_1 x^1 + \cdots + a_n x^n$. The primitive functions $x^1, x^2, \ldots, x^n$ can be computed at different nodes of a network. Each one of them has to be connected to the single input x through an edge with weight 1. The results of each node go to an output unit through $n + 1$ edges with weights $a_0, a_1, \ldots, a_n$. The optimal weights for this very simple network can be found applying the least squares method to the training set. The same kind of construction can be used to approximate a given training set by additions of sines and cosines, i.e. by a Fourier series.

Multilayer networks are a generalization of all these kind of approaches to the approximation of functions, because we consider not only linear combinations of primitive functions (like with polynomials or Fourier series), but also almost any kind of function composition. We thus seek to approximate unknown functions with a more richer set of combinatorial possibilities. By

accepting also nonlinear combinations of primitive functions, we can do better as with simple polynomial approximations, which are now just a subcase of the whole method. This increased power is unfortunately the main reason for the slow convergence of the known learning algorithms. It has been shown that learning in neural networks is an NP-complete problem, that is one for which most probably no algorithm with polynomial time complexity in the number of weights exist [Jud87].

Backpropagation is thus in some sense a generalization of the least-squares method used for approximating functions through polynomials. After the network weights have been found using the algorithm, the network is tested with unknown input patterns and we expect it to be able to *generalize*, that means, it should map neighborhoods of the known input patterns into neighborhoods of the associated output patterns. Figure 2 shows an example in which the 16×16 square was divided in three clusters of squares. The shading values of 120 pixels chosen at random were learnt by a network with 8 input, 5 hidden and 3 output units. The Figure shows the actual shading pattern learnt by the network. The result is reasonable good also for points not used in the training step.

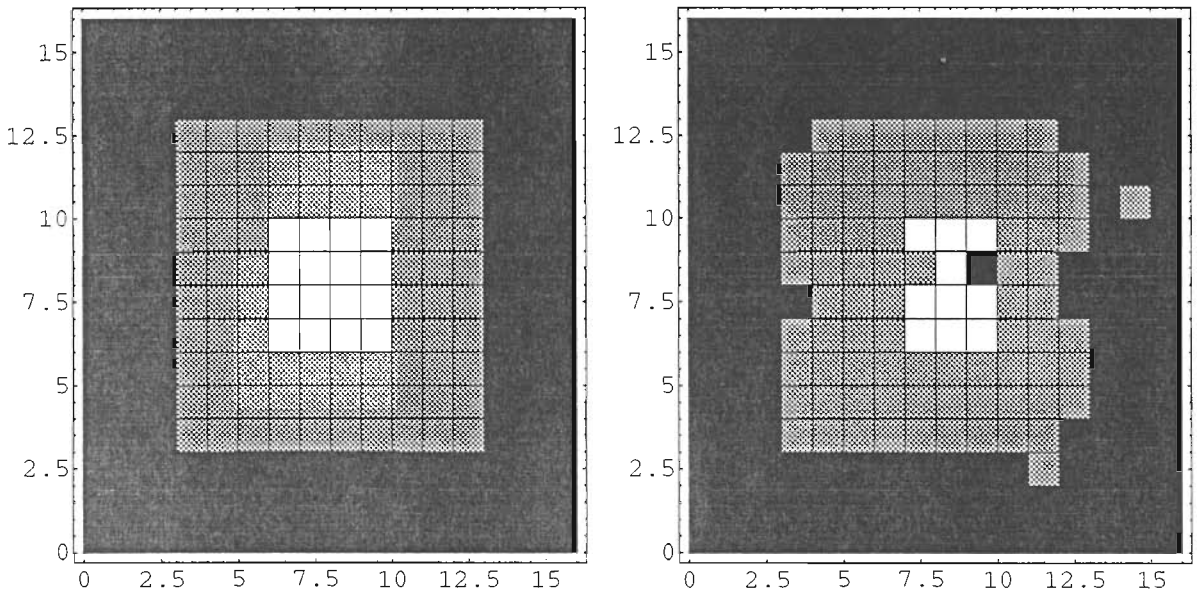


Figure 2: Original clustering and network's classification.

This paper provides an overview of the vast field of backpropagation algorithms and the problems associated with training multilayer neural networks. After a review of the basics of the backpropagation algorithm, we look at some of its most important variations. We have divided them in different classes according to their basic approach. We discuss the inner workings of each algorithm, its advantages and drawbacks as well as how it is related to other methods.

Since neural networks are massively parallel learning models, we concentrated our attention on learning algorithms that preserve the network flow computational model. Some researchers have proposed algorithms, which they claim are superior to standard backpropagation, but that use weight-updating strategies based on non-local information. For all backpropagation variations

examined in this paper, the neurons need no more information than the one they can get through their input- or output channels. No global information is involved.
After the discussion of the learning algorithms, we describe the conditions under which we tested them. This includes a brief review of possible benchmarks, and a description of the difficulties involved in choosing the right ones. Another important question is 'When is learning complete?' [Fah88]. We consider a variety of possible stopping criterias, which depend on the problem to be learned. We also give some suggestions for the detection of local minima.
Last but not least, a run-time comparison of the algorithms on the basis of selected benchmarks is made. This includes an estimation of the 'optimal' parameters of standard backpropagation for each benchmark and the comparison of this 'optimal standard' algorithm with other backpropagation variations.

2 Learning in Neural Networks

Consider a feed-forward network with n input and m output units. It can consist of any number of units and can exhibit any desired feed-forward connection pattern. We are also given a training set $(x_1, t_1), (x_2, t_2), \ldots, (x_p, t_p)$ consisting of p ordered pairs of n- and m-dimensional vectors, which are called the *input* and *output patterns* respectively. Let the primitive functions calculated 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 x_i from the training set is presented to this network, it produces an output o_i different in general from t_i. What we want. is to make o_i and 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

$$E(W) := \sum_{i=1}^{i} E_i = \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{m} (o_{ij} - t_{ij})^2. \tag{1}$$

The first step of the minimization process consists in extending the network, so that it computes the error function automatically. Figure 3 shows how this is done. Every output unit $j = 1, \ldots, m$ 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 o_i, respectively the target t_i. The output of the new m nodes is collected at a node which just adds them up and gives the sum E_i, that is the quadratic error for the input pattern p_i as its output. The same extension has to be done for each pattern t_i. An additional computing unit collects all quadratic errors and outputs their sum. The output of this extended network is of course the error function $E(W)$.

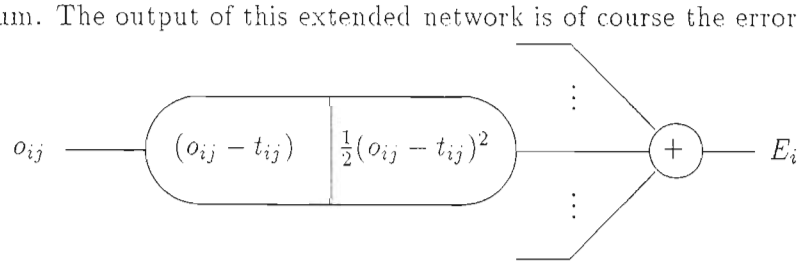


Figure 3: Extension of the network for the computation of the error function.

We have now a network capable of calculating the error function for a given training set. The weights in the network are the only parameters that can be changed. We can tune them, trying

to make the quadratic error E as low as possible. Since 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 $w_1, w_2, \ldots, w_l$ of the network. We can thus minimize E by using an iterative process of gradient descent for which we need to calculate the gradient

$$\nabla E(W) = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots \frac{\partial E}{\partial w_l});$$

we then adjust each weight w_i by using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i}; \quad i = 1, \ldots, l,$$

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 reduces now 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 $\nabla E(W) = 0$.

2.1 The Backpropagation Algorithm

Forget now everything about training sets and learning. Our objective is just to find a method for calculating efficiently 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 consists now of a left and a right side. 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. The network is evaluated now in two stages: in the first one, the *feed-forward step*, information comes from the right and each unit evaluates its primitive function f in its right side as well as the derivative of f in its left side. Both results are stored in the unit, but only the result from the right side is transmitted to the next node. Now we have to distinguish the following cases:

2.1.1 Function Composition

The network of Figure 4 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. Inthis step, the network computes the composition of the functions f and g. The Figure shows the state of each unit after the feed-forward step. The correct result of the function composition has been produced at the output neuron and each neuron has stored some information in 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* with the value stored in its left side. The result of the multiplication is transmitted to the left and the information is transported to the next unit. We call the result at each node the *traversing value* at this node. Figure 4 shows the final result of the backpropagation step, which is $f'(g(x))g'(x)$, that is the derivative of the function

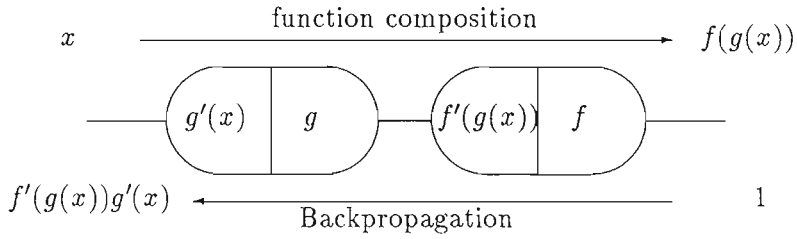


Figure 4: Backpropagation for the composition of two functions.

composition $f \circ g$ implemented by this network. The backpropagation step provides us thus with 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.

2.1.2 Function Addition

The next case to consider is the addition of two primitive functions. Figure 5 shows a network to compute the addition of the functions f_1 and f_2. The additional node has been included just to handle the addition of the two functions. Its activation function is the identity, whose derivative 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 network fan-out the traversing value at this node and distribute it to the connected neurons. When two paths meet, the computed traversing values are added. Figure 5 shows the result of the backpropagation step for the network. The result is the derivative of the function addition $f_1 + f_2$. A simple proof by induction shows that the derivative of the addition of any number of functions can be handled in the same way.

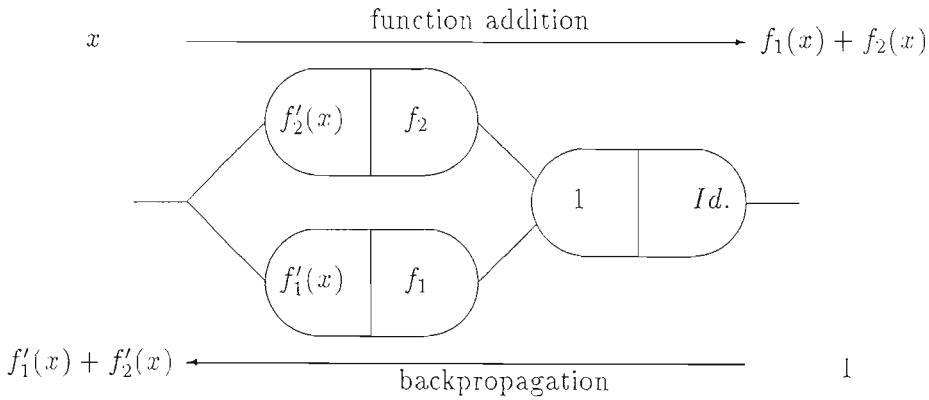


Figure 5: Backpropagation for the addition of two functions.

2.1.3 Weighted Edges

The last case we have to consider are weighted edges. In the feed-forward step the incoming information x is multiplied with the edge's weight w. The result is wx. In the backpropagation step the traversing value 1 is multiplied with the weight of the edge. The result is w, which is the derivative of wx with respect to x. We conclude from this case that weighted edges are used exactly in the same way in both steps: they modulate the information transmitted in each direction by multiplying it with the edges weights.

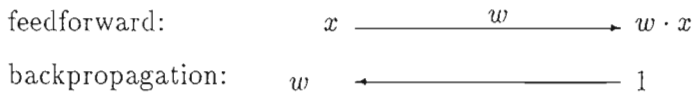


Figure 6: Backpropagation at an edge.

We can now formulate the complete backpropagation algorithm and give a proof 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. The two phases of the algorithm are the following:

1. **Feed-forward step:** 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.

2. **Backpropagation step:** The constant 1 is fed into the output unit and the network is run backwards. All incoming information to a node is added and the result is multiplied with 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.

We showed before that the algorithm works for units in series, units in parallel and also when weighted edges are present. Let us make the induction assumption that the algorithm works for any feed-forward network with n or less that n nodes. Consider now the network of Figure 7, which is made of $n+1$ nodes.
The feed-forward step is executed first and the result at the output unit is the network function F evaluated at x. Assume that m units, whose respective outputs are $F_1(x), F_2(x), \ldots, F_m(x)$, are connected to the output unit. Since the primitive function of the output unit is ϕ, we know that

$$F(x) = \phi(F_1(x) + F_2(x), + \cdots + F_m(x)).$$

The derivative of F at x is thus

$$F'(x) = \phi'(s)(F_1'(x) + F_2'(x), + \cdots + F_m'(x)). \tag{2}$$

where $s := F_1(x) + F_2(x), + \cdots + F_m(x)$. 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 less units. By the induction assumption we can

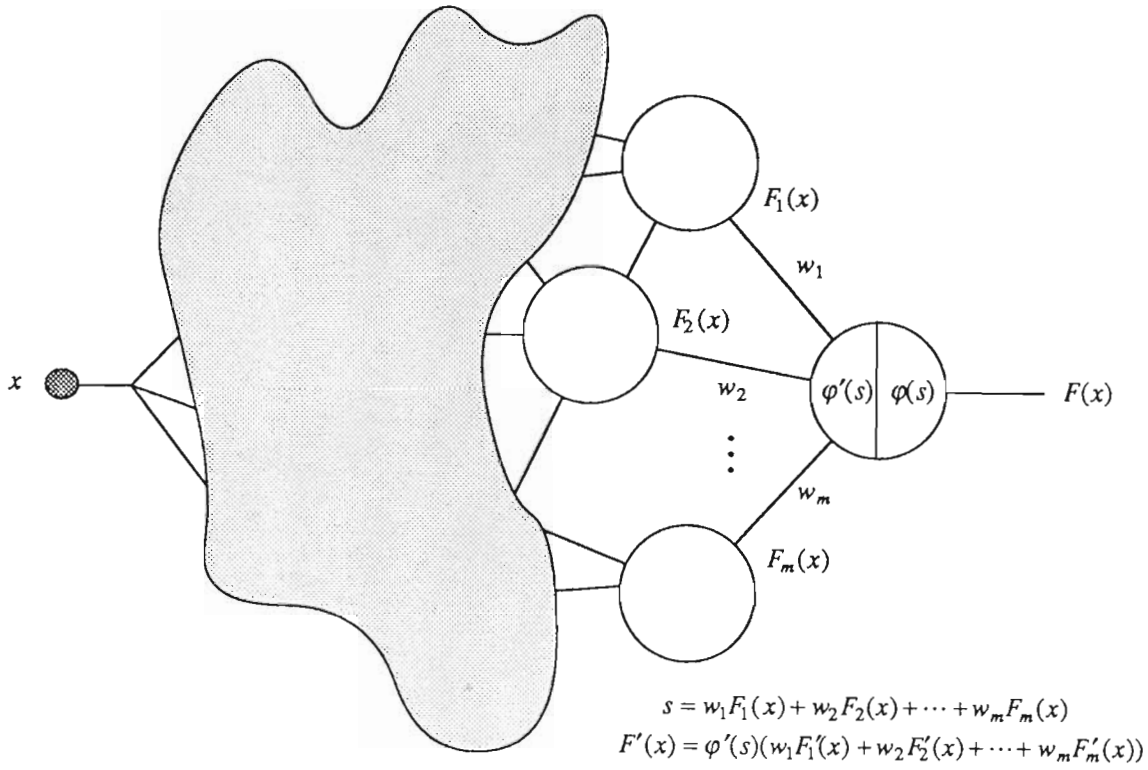


Figure 7: Backpropagation at the last node.

calculate the derivative of F_1 at x, by injecting a 1 into the unit and running the subnetwork backwards. The same can be done with the units whose output is $F_2(x), \ldots, F_m(x)$. If we inject the constant $\phi'(s)w_1$ instead of a 1, we get at the input unit in the backpropagation step $w_1F_1'(x)\phi'(s)$ and $w_2F_2'(x)\phi'(s), \ldots, w_mF_m'(x)\phi'(s)$ with the rest of the units. But in the backpropagation step with the whole network we add these m results and we get (2), which is the derivative of F evaluated at x. Note that injecting the constants $w_1F_1'(x)\phi'(s), \ldots, w_mF_m'(x)\phi'(s)$ into the m units connected to the output unit, can be done by injecting a 1 in the output unit, multiplying with the stored value $\phi'(s)$ 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 just like the backpropagation algorithm demands. This means that the algorithm works with networks of $n+1$ nodes and this concludes our proof.

The backpropagation algorithm still 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 and y are fed into the network, the network result can be called $F(x, y)$. The network function has now two arguments and we can compute the partial derivative of F with respect to x or with respect to y. The feed-forward step remains unchanged and all left-side slots of the units are filled as usual. But 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 at the first input unit. The backpropagation step on the second subnetwork yields the partial derivative of F with respect to y 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.

2.2 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 each weight in the network at a time.

The feed-forward step for the l-th input pattern is computed in the usual way, but we now also store the output of each neuron in its right side. We perform the backpropagation step in the extended network used to compute the error function and we then fix out attention in one of the weights, say w_{ij} which points from the i-th to the 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_l with respect to this input, that is $\partial E_l / \partial o_i w_{ij}$. Since in the backpropagation step o_i is treated as a constant, we finally have

$$\frac{\partial E_l}{\partial w_{ij}} = o_i \frac{\partial E_l}{\partial o_i w_{ij}} \tag{3}$$

for $o_i \neq 0$. In case that $o_i = 0$ then $\partial E_l / \partial w_{ij}$ is just zero. 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 a third quantity at each node: the 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 write the partial derivative of E_l with respect to w_{ij} as

$$\frac{\partial E_l}{\partial w_{ij}} = o_i \delta_i$$

The partial derivative of E with respect to w_{ij} is the sum of all this partial derivatives for $l = 1, \ldots, p$. Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$$\Delta w_{ij} = -\gamma \frac{\partial E}{\partial w_{ij}}.$$

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

Let's consider how backpropagation works in a neural network with a *layered* architecture, like the one shown in Figure 1. We assume, that the network consits of one layer of input units, one layer of hidden units and one layer of output units with no direct connection between input-

and output layer. This is the most common architecture. All neurons use the *sigmoid function* $s(x)$ as their activation function, where $s(x)$ is defined as

$$s(x) := \frac{1}{1+e^{-x}}. \tag{4}$$

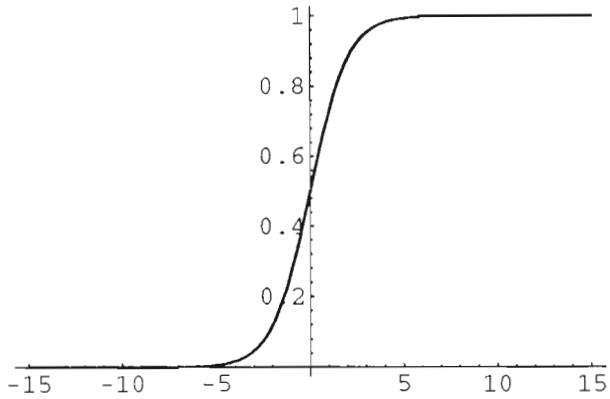


Figure 8: Graph of the sigmoid activation function.

The first derivative of the sigmoid is

$$\frac{\partial s(x)}{\partial x} = s(x)(1-s(x)). \tag{5}$$

We assume further, that there is only one pattern to be learned. This is not really a restriction, it just makes the algorithm easier to understand. The case of more than one pattern to learn is described in section 3.1.1.
Following notation holds from now on:

$w_{ij}^{(1)}$	:	Weights between component i of the input and neuron j of the hidden layer.
$w_{ij}^{(2)}$	:	Weights between neuron i of the hidden layer and neuron j of the output layer.
$o_i^{(0)}$	:	i-th component of the input vector.
$o_i^{(1)}$	:	Output of neuron i of the hidden layer.
$o_i^{(2)}$	:	Output of neuron i of the output layer.
n_I	:	Dimension of the input layer L_I.
n_H	:	Dimension of the hidden layer L_H.
n_O	:	Dimension of the output layer L_O.

We assume that the feedforward computation has already been done, that is some input vector $o^{(0)}$ was fed into the network and the output $o^{(2)}$ was computed. All the partial derivatives of the activation functions have also been computed and stored in the left sides of the nodes. The

output of the network output has been compared to the desired output t, and the error function $E(W)$ has been computed by the additional layer. Now we can perform what gave its name to the algorithm: The propagation of the error back through the network.
We start with the value $T = 1$ on the right hand side and go back to the output layer. After going through neuron j of the additional layer, we get a new value for T:

$$T := (t_j - o_j^{(2)}), \qquad j = 1, \ldots, n_O,$$

since this is the derivative of the error function with respect to $o_j^{(2)}$. We now go through neuron j of the output layer, which is the only neuron in the output layer, that is connected to neuron j of the additional layer. If we leave this neuron via the connection weighted by $w_{ij}^{(2)}$ in the direction of neuron i of the hidden layer, we finally get

$$T := (t_j - o_j^{(2)})o_j^{(2)}(1 - o_j^{(2)})o_i^{(1)} \qquad i = 1, \ldots, n_H; \quad j = 1, \ldots, n_O,$$

an this is the partial derivative of the error function with respect to $w_{ij}^{(2)}$. So with

$$\delta_j^{(2)} := -(t_j - o_j^{(2)})o_j^{(2)}(1 - o_j^{(2)}) \qquad j = 1, \ldots, n_O \tag{6}$$

we get

$$\frac{\partial E}{\partial w_{ij}^{(2)}} = -\delta_j^{(2)} o_i^{(1)} \qquad i = 1, \ldots, n_H; \quad j = 1, \ldots, n_O. \tag{7}$$

Propagating the error further backwards through the network we can compute the derivatives $\partial E / \partial w_{ij}^{(1)}$. In this case we get

$$\delta_j^{(1)} := (\sum_{l=1}^{n_H} \delta_l^{(2)} w_{lj}^{(2)}) o_j^{(1)}(1 - o_j^{(1)}) \qquad j = 1, \ldots, n_H$$

and

$$\frac{\partial E}{\partial w_{ij}^{(1)}} = -\delta_j^{(1)} o_i^{(0)} \qquad i = 1, \ldots, n_I; \quad j = 1, \ldots, n_H. \tag{8}$$

In conclusion, learning with the backpropagation algorithm in multilayered networks consists of the following steps:

1. **Feedforward-step.** The input vector $o^{(0)} = (o_1^{(0)}, o_2^{(0)}, \ldots, o_{n_I}^{(0)})$ is fed into the network, which computes the output $o^{(2)}$. The additional layer estimates the error function $E(W)$.

2. **Backpropagation-step to the output layer.** The error is propagated back through the network to compute the partial derivatives $\partial E / \partial w_{ij}^{(2)} = -\delta_j^{(2)} o_i^{(1)}$.

3. **Backpropagation-step to the hidden layer.** The error is propagated further back through the network to compute the partial derivatives $\partial E / \partial w_{ij}^{(1)} = -\delta_j^{(1)} o_i^{(0)}$.

4. **Error correction step.** After the evaluation of all partial derivatives the weights are corrected in the direction of the negative gradient, which means

$$\begin{aligned} \Delta w_{ij}^{(2)} &= \gamma \delta_j^{(2)} o_i^{(1)} \qquad i = 1, \ldots, n_H; \quad j = 1, \ldots, n_O \\ \Delta w_{ij}^{(1)} &= \gamma \delta_j^{(1)} o_i^{(0)} \qquad i = 1, \ldots, n_I; \quad j = 1, \ldots, n_H, \end{aligned} \tag{9}$$

where $\gamma > 0$ is a learning parameter. Some researchers have proposed different learning parameters for the different layers, but for the standard algorithm, we always used the same constant learning rate for all layers.

It is very important, first to compute *all* the partial derivatives before the error correction (9) is performed. Otherwise the computation of the gradient is mixed up with the correction of the weights, and we can get a wrong result for the gradient.

2.3 The 'Pros and Contras' of Backpropagation

As we have seen, backpropagation is a learning procedure for multilayer perceptrons, which is basically a gradient descent method. It is very easy to implement and it is also highly parallel, since it needs no global information to run.
But since it is a gradient descent algorithm, it also has all the drawbacks of such methods. The algorithm converges very slowly and it also can get stuck in local minima or flat plateaus of the error function very easily. Another difficulty is the choice of the learning parameter γ (see (9)). If it is chosen very small, the algorithm will almost follow the optimal gradient direction (which does not mean, that this does not lead to a local minimum), but it will take many iterations, especially if the error function is very 'flat'. On the other hand, if γ is chosen too big, the algorithm may 'jump' over some valleys, maybe the one we look for. The choice of γ always depends on the shape of the error function, which makes it very difficult, since usually we know nothing about it.

3 Accelerating the Backpropagation Algorithm

In this section, various approches to improve backpropagation will be described. We divided them in the following classes:

1. Standard-Variations (Sec. 3.1)

2. Adaptive-Step Algorithms (Sec. 3.2)

3. Second-Order Algorithms (Sec. 3.3)

There is of course no sharp distinction between these classes, they are all more or less related. These connections, the basic idea of the algorithms in each class, as well as their most important representatives are now described in deep.

3.1 Variations of the Standard Algorithm

Before we get to the more sophisticated methods, we would like to describe some basic modifications of the standard algorithm, which may even be useful to speed the other algorithms up.

3.1.1 More than one pattern to learn: Batching vs. On-Line

The first decision to be made is how to update the weights, if there is more than one pattern to learn. We usually have two choices:

- **Batching:** All the training patterns are passed through the network one after the other, the partial corrections Δw_{ij} are accumulated, and the error correction takes place at the end, when all the patterns have been presented to the network. This procedure estimates the true gradient, but with a big computational effort, since the weights are only updated once after every presentation of the 'batch' of training patterns.
- **On-line backpropagation:** This method, also called **real time** backpropagation, presents again the training patterns to the network one after the other, but the correction of the weights takes place after each single pattern has been shown to the network. It is better to choose the patterns to be learnt by random, rather than to pick them one after the other. Updating of the weights is not done exactly in the direction of the true gradient, the corrections made oscillate around the true gradient. With on-line backpropagation, the stepsize γ should not be chosen too big, otherwise the learning procedure may be 'misleaded'.

Experiments show, that backpropagation converges faster, if the corrections are done in the direction of the true gradient estimated by batching, even if there are, with the same computational effort, less weight updates per iteration than with on-line backpropagation. This holds especially for more sophisticated algorithms, which implement large steps in weight space, trying to find a solution in very few iterations.
If the training set contains a very large number (say hundreds) of patterns to learn, one can make a compromise and update the weights after the presentation of a subset (let's say 50) of the training patterns selected at random. Otherwise the iterations get too expensive, and often a subset, which is not too small, contains all the information needed to make a reasonable estimation of the gradient.
Methods that try to speed up backpropagation by replacing the learning set by a much smaller set (not necessarily a subset of the training set) of vectors, that still contains much of the information of the whole learning set are described in section 4.

3.1.2 Introduction of a Momentum Term

Rumelhart noted, that backpropagation is quite slow, if the learning rate γ is small, but it may lead to oscillations, if one is trying to speed it up by chosing a bigger learning rate. His suggestion [RHW86] aimed at increasing the learning rate while simultaneously avoiding these oscilations, was to introduce a *momentum term* into the error correction, modifying (9) to

$$\Delta W^{(k)} := -\gamma \nabla E(W^{(k)}) + \alpha \Delta W^{(k-1)}. \tag{10}$$

where k denotes the current iteration and α a new real parameter. The advantage of using such a momentum term is twofold. First, if the computed gradient $-\nabla E(W^{(k)})$ and the correction $\Delta W^{(k-1)}$ from the last step point in similar directions, the step size of the actual correction is increased. This leads to a speed-up in rather 'flat' parts of the error function. Second, if we have some 'narrow valley' to climb down, we may get some annoying oscillations, which could be eliminated by using the momentum term. Figure 9 shows an example.

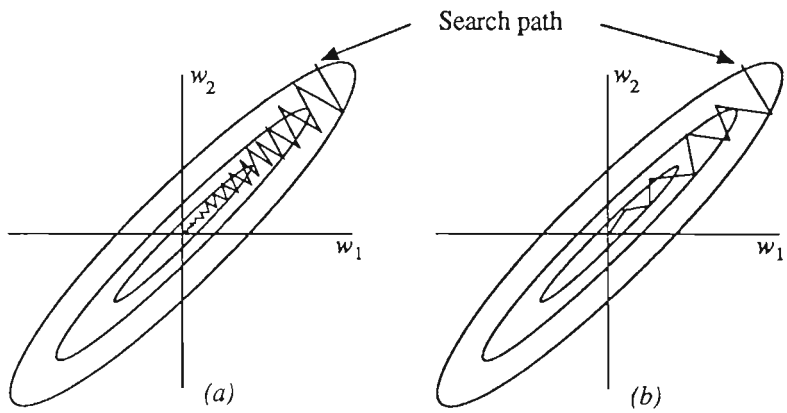


Figure 9: Descending a narrow valley with and without the use of a momentum term.

Although of course α has to be chosen empirically, just like γ, we can say that since

$$\Delta W^{(k)} = -\gamma \nabla E(W^{(k)}) + \alpha \Delta W^{(k-1)} = -\gamma \nabla E(W^{(k)}) - \gamma \sum_{j=1}^{k-1} \alpha^j \nabla E(W^{(k-j)}) + \alpha^k \Delta W^{(0)}$$

$\alpha \leq 1$ has to hold, otherwise the momentum term (especially $\alpha^k \Delta W^{(0)}$) explodes, as k increases. Since the momentum term is pulling us away from the steepest descent direction $-\nabla E(W^{(k)})$, we need a rather good estimation of the gradient, if we do not want to get hopelessly lost on the surface of the error function. Thus, batching is recommended, to obtain the true gradient of the error function in every iteration.
One correction according to (10) now consists of the use of the actual gradient $\nabla E(W^{(k)})$ and some linear combination of the gradients computed during the steps before. This relates backpropagation with momentum to the cg methods. The difference is, that cg methods use mutually conjugate search directions, which does not necessarily hold for the $\nabla E(W^{(j)})$, $j = 1, \ldots k$. A rather extensive analytical study of the momentum term has been done by A. Sato in [Sat91].

3.1.3 Using bipolar- instead of binary vectors

Another simple approach is to train the network with bipolar vectors, which consist of elements equal to -1 or 1. This means, that every component x_i of the binary vector x is replaced by the element $\hat{x}_i := 2x_i - 1$ to obtain the bipolar vector $\hat{x}$. The activation function of the neuron has to be changed then from the sigmoid $s(x)$ to the symmetric sigmoid $\hat{s}(x) := 2s(x) - 1$.

Two arbitrary bipolar vectors are orthogonal, and thus decorrelated, with a probability, that increases with the dimension n of the vectors. To see this, we have to look at the scalar product

$$S = \sum_{k=1}^{n} \hat{x}_k \hat{y}_k$$

of two arbitrary binary vectors $\hat{x}$ and $\hat{y}$. The single components $\hat{x}_k \hat{y}_k$ are either -1 or 1, both with probability 0.5. For increasing n, the probability of $S = 0$ approaches one. For binary vectors on the other hand, $S \approx n/4$, because $x_k y_k$ are either 0, with probability 0.75 or 1, with probability 0.25.
This is a reason for the speedup of backpropagation when bipolar vectors are used, because as described in section 4, an orthonormalized and decorrelated data set has positive effects on the error function.
Another, much simpler positive effect of using bipolar vectors can be understood, if we take a look at the error correction step (9) of the backpropagation algorithm. It shows that the weight's corrections $\Delta w_{ij}^{(1)}$ are proportional to the input $o_i^{(0)}$. So the weights $w_{ij}^{(1)}$ are only changed for $o_i^{(0)} \neq 0$, which always holds for bipolar vectors. This has already been pointed out by W. S. Stornetta and B. A. Hubermann [SH87] and has also bee used by S. E. Fahlmann [Fah88], who used 'constrained bipolar' vectors consisting of elements equal to -1/2 and 1/2 and a symmetric sigmoid $\hat{s}(x) := s(x) - 1/2$. This still has the positive effects described above, but it does not make the activation function steeper and we also save a multiplication at each node.

3.1.4 Introduction of a temperature parameter in the sigmoid

The *temperature* is a constant $c > 1$ which acts multiplicatively on the input of the sigmoid, changing the activation function to

$$s(x) := \frac{1}{1 + e^{-cx}}$$

and its first derivative to

$$\frac{\partial s(x)}{\partial x} = c \cdot s(x)(1 - s(x)).$$

This has the effect of making the step of the sigmoid (and simultaneously, the steps of the error function) steeper. Thus fewer iterations are required to climb it down. *Simulated annealing* is a technique which uses a changing temperature constant and is used in stochastic neuronal networks, such as *Boltzmann Machines*.
Although the networks learn somewhat faster in some small problems, like the XOR function, this technique will in general rather lead to wild oscillations of the network's weights, since it basically works just as an increment of the learning rate γ. Experiments also show, that the introduction of a temperature constant has a negative effect on the generalization abilities of the network. The explanation is that because of the steeper step of the sigmoid, the neurons 'flip' their output faster. Unknown inputs, near to learned ones, may not produce the desired output, which should be near to the output belonging to the learned input.
After some experiments we came to the conclusion, that the negative effects of introducing the temperature constant overcome the positive ones, so we did not look further at this technique.

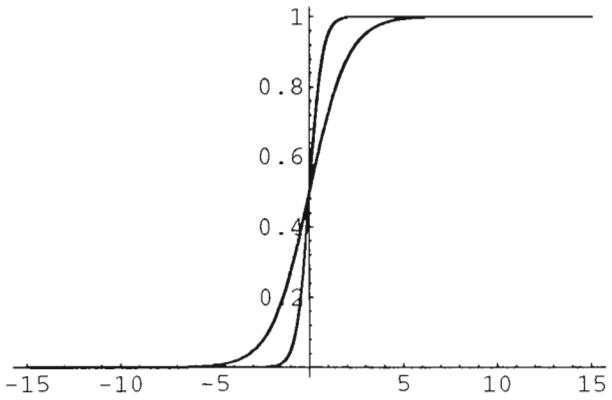


Figure 10: The Sigmoid function for $c = 1$ and $c = 3$.

3.1.5 Handling flat spots of the error function

The first derivative of the sigmoid (5) goes to zero, as the output of the neuron goes to zero or one. Since the correction of the weights is proportional to these derivatives, they will then also go to zero. These regions of the error function are called the *flat spots* of the error surface. The presence of these flat spots is one of the main reasons for the slow convergence of standard backpropagation. Several proposals have been made to solve this problem.

1. **Adding an offset to the derivative of the sigmoid**
   S. E. Fahlmann [Fah88] proposed making sure, that the sigmoid's derivative will never get too close to zero. His idea was to add a small constant $\epsilon > 0$ to the first derivative of the sigmoid,

   $$\frac{\partial s(x)}{\partial x} + \epsilon = s(x)(1 - s(x)) + \epsilon, \tag{11}$$

   so that this expression is always greater than zero (at least ϵ) and the weight's changes will not die out. Good results were obtained with $\epsilon \approx 0.1$.

2. **Modification of the error function I**
   K. Balakrishnan and V. Honavar [BH92] proposed a modification of the error function, to eliminate at least the flat spots in the output layer. Their approach is not to evaluate the error between the *output* of the output neurons $o_{pi}^{(2)}$ and the desired output t_{pi}, but between the *input* of the output neurons and their desired input, which is easy to compute as $s^{-1}(t_{pi})$. This changes the error function to

   $$E = \sum_p E_p = \frac{1}{2} \sum_p \sum_i (netTarget_{pi} - netIn_{pi})^2, \tag{12}$$

where $netTarget_i$ is the desired input of output neuron i for pattern p and $netIn_{pi}$ is its actual input. We can now evaluate the weight update rules just as in section 2.2 with the only difference, that $\delta_i^{(2)}$ becomes

$$\delta_i^{(2)} := (netTarget_{pi} - netIn_{pi}).$$

The sigmoid is now approximated lineary to obtain an approximation of its derivative

$$s' = \frac{outputError_{pi}}{inputError_{pi}} = \frac{t_i - o_{pi}^{(2)}}{netTarget_{pi} - netIn_{pi}}.$$

Now $\hat{\delta}_i^{(2)}$ becomes

$$\hat{\delta}_i^{(2)} := \frac{target_{pi} - o_{pi}^{(2)}}{s'} \tag{13}$$

To make sure, that s' will not get too close to zero, Balakrishnan and Honavar define an output unit to be in the *active range* if its output $o_j^{(2)}$ is greater than $(1-m)$ or less than m (where they chose $m = 0.9$) and in the *inactive range* otherwise. If an output unit is found to be in the active range, the updates are made using $\hat{\delta}_i^{(2)}$ of (13), otherwise s' is replaced by a constant value of 0.09.

Since s' is always smaller than 0.25 and can get close to zero, the weight updates are divided by 10.0 to make sure, that they will not become too large, because this could lead to oscilations of the networks weights.

3. **Modification of the Error Function II**
A. Kryzak, W. Dai and C. Y. Sun proposed another modification to handle flat spots in the output layer [KDS90]. They define the error E_{pj} of each output neuron j for a certain pattern t_p as

$$E_{pj} := \begin{cases} -(1-t_{pj})ln(1-o_{pj}^{(2)}) & if \quad t_{pj} = 0 \\ -t_{pj}ln(o_{pj}^{(2)}) & if \quad t_{pj} = 1 \end{cases}$$

This changes the error function and its derivative to

$$\begin{aligned} E_p &= \sum_j E_{pj} = -\sum_j((1-t_{pj})ln(1-o_{pj}^{(2)}) + t_{pj}ln(o_{pj}^{(2)})) \\ \frac{\partial E_p}{\partial o_{pj}^{(2)}} &= -t_{pj}\frac{1}{o_{pj}^{(2)}} + (1-t_{pj})\frac{1}{1-o_{pj}^{(2)}}. \end{aligned}$$

This leads to a redefinition of $\delta_j^{(2)}$ (see (6))

$$\hat{\delta_j^{(2)}} = -(-t_{pj}\frac{1}{o_{pj}^{(2)}} + (1-t_{pj})\frac{1}{1-o_{pj}^{(2)}})o_{pj}^{(2)}(1-o_{pj}^{(2)})) = t_{pj} - o_{pj}^{(2)}, \tag{14}$$

where $\delta_j^{(1)}$ remains unchanged. Since this approach iis undefined for nonbinary output, we dropped this error function modification for the comparison.

3.2 Adaptive-step algorithms

The idea behind this kind of methods is to use *variable step sizes* instead of a constant learning rate γ for the corrections defined by (9). This step size is often chosen to be a function of the backpropagated error. This means that for steep descent, where the backpropagated error is big and thus significant changes of the error function $E(W)$ are expected, γ is chosen to be small to avoid overjumping the minimum; where the backpropagated error is small and the error function is rather flat, γ is chosen to be big, so that not too many steps are needed to reach the minimum. Such an adptive rule is of course a heuristic and depends highly on the problem to be learned. Some of these methods tune the learning parameter used in every step by using information gained during the learning process.

3.2.1 The gradient reuse algorithm

This algorithm, proposed by D. R. Hush and J. M. Salas [HS88] is surely the most obvious approach for this tree of acceleration techniques. It is actually a kind of line search. The search direction is given by $-\nabla E(W^{(k)})$ and the minimum (or some point close to it) along this direction is to be found. There is of course a trade-off between accuracy (getting as close as possible to the minimum) and computational effort (checking only a few points along the search direction).

The search direction $-\nabla E(W^{(k)})$ is followed in discrete steps $W^{(k_{i+1})} := W^{(k_i)} - \gamma^{(k)}\nabla E(W^{(k)})$, as long as the error function decreases. If $E(W^{(k_{i+1})}) > E(W^{(k_i)})$ holds, $W^{(k+1)} := W^{(k_i)}$ is declared to be the minimum along that direction. A new search direction $-\nabla E(W^{(k+1)})$ is computed, and the algorithm continues.

The stepsize $\gamma^{(k)}$ depends again on the behavior of the error function. If in step $k-1$ the gradient $-\nabla E(W^{(k)})$ was reused very often, the error function is assumed to be flat and $\gamma^{(k)}$ is increased, to obtain a smaller reuse rate and thus to investigate fewer points along the search direction. If, on the other hand, the reuse rate during step $k-1$ was low, the error function is assumed to be rather steep and $\gamma^{(k)}$ is decreased to make sure that the minimum is approximated sufficiently well. Hush and Salas claim that a reuse rate of about ten iterations is the best.

Since the gradient is reused several times, we need a rather accurate search direction. Thus, batching is used to obtain a better result for $\nabla E(W)$.

3.2.2 The dynamic adaption algorithm

R. Salomon suggested a similar procedure, but taking only two points along the search direction and adjusting the learning parameter $\gamma^{(k)}$ dynamically [Sal92]. This means that for a given search direction $d = \nabla E(W)$ and a given learning parameter $\gamma^{(k)}$ the points $W^{(k-1)} - d \cdot \gamma^{(k)}\zeta$ and $W^{(k-1)} - d \cdot \gamma^{(k)}/\zeta$ are examined. The point $W^{(k+1)}$, that causes the smallest error and the corresponding new learning rate are chosen. We have the following equations:

$$
\begin{aligned}
E(W^{(k-1)}, \gamma^{(k)}) &= E(W^{(k-1)} - \gamma^{(k)}\nabla E(W^{(k-1)})) \\
\gamma^{(k)} &= \begin{cases} \gamma^{(k-1)} \cdot \zeta & \text{if } E(W^{(k-1)}, \zeta\gamma^{(k-1)}) \leq E(W^{(k-1)}, \frac{1}{\zeta}\gamma^{(k-1)}) \\ \gamma^{(k-1)} \cdot \frac{1}{\zeta} & \text{otherwise} \end{cases} \\
W^{(k)} &= W^{(k-1)} - \gamma^{(k)}\nabla E(W^{(k)})
\end{aligned}
$$

He sets $\zeta = 1.3$, and so did we. In the original version Salomon normalizes the gradient $\nabla E(W)$, but claims that the algorithm above also converges without this normalisation, only more slowly. We did not normalize the gradient because of locality reasons.

3.2.3 The Delta-Bar-Delta algorithm

This algorithm, developed by R. A. Jacobs [Jac88], uses different learning rates for every single weight, which are adapted at each iteration. This approach reflects the idea that the slope of the error surface might differ considerably, depending on the weight directions. Since backpropagation corrections with a constant learning rate are proportional to these slopes (3),(8), the size of the step actually taken in weight space may also differ considerably. This may lead, for steep descent and thus large slopes, to minima being jumped over. On the other hand, it may lead to very slow descents at flat spots with small slopes. To solve these problems, Jacobs suggested the following heuristics:

1. Each weight (each direction in the search space) has its own learning rate.
2. These learning rates are modified based on information about the error surface .
3. When the error gradient $\partial E/\partial w_{ij}$ has the same sign in many consecutive iterations, the corresponding *learning rate is increased*, since a minimum may lie ahead.
4. When the gradient flips signs in several consecutive iterations, the *learning rate is decreased*, since this indicates, that a minimum has been overjumped.

These heuristics, which form the basis of the Delta-Bar-Delta algorithm, lead to the following equations, where k indicates the iteration number.

$$\begin{aligned}
\gamma_{ij}^{(k+1)} &= \gamma_{ij}^{(k)} + \Delta\gamma_{ij}^{(k)} \\
\Delta\gamma_{ij}^{(k)} &= \begin{cases} \kappa & \text{if } \hat{S}_{ij}^{(k-1)}\nabla E(W^{(k)}) > 0 \\ -\phi\gamma_{ij}^{(k)} & \text{if } \hat{S}_{ij}^{(k-1)}\nabla E(W^{(k)}) < 0 \\ 0 & \text{otherwise} \end{cases} \\
\hat{S}_{ij}^{(k)} &= (1-\theta)\nabla E(W^{(k)}) + \theta\hat{S}_{ij}^{(k-1)}
\end{aligned} \tag{15}$$

The term $\hat{S}_{ij}^{(k)}$ is basically a decaying trace of gradient values. The parameters κ, ϕ and θ are to be specified by the user. Experiments show, that values of $\kappa \approx 0.05$, $\phi \approx 0.3$ and $\theta \approx 0.7$ work best [Jac88, MW90]. The learning rates $\gamma_{ij}^{(0)}$ may be set to the initial values 0.1.
The reason for increasing the learning rates additively is to prevent them from becoming too large too fast; the reason for decreasing them exponentially is to keep them positive at all iterations, as well as to allow rapid decreases.

3.2.4 The extended Delta-Bar-Delta algorithm

A. A. Minai and R. D. Williams found that the Delta-Bar-Delta algorithm has several drawbacks [MW90]. First, the introduction of a momentum term, which is a rather elegant method

of speeding standard backpropagation up, sometimes causes the Delta-Bar-Delta algorithm to diverge. Second, even with very small values κ (see (15)), the learning rates can increase so much, that the small exponential decreases do not prevent the algorithm from jumping wildly. To overcome these drawbacks, Minai and Williams suggested the following modifications to the Delta-Bar-Delta algorithm:

1. The learning rate increase is changed from a constant increase κ to an exponentially increasing function of $|\hat{S}_{ij}^{(k)}|$. This causes the learning rate to increase faster on flat spots of the error surface than on rather deep slopes.

2. Momentum is used as a standard part of the algorithm, and the momentum is changed adaptively just like the learning rate, using the Delta-Bar-Delta criteria.

3. To prevent the learning and momentum rate from becoming too large, an upper limit is defined for both.

4. *Memory* and *recovery* are incorporated into the algorithm, which means that the best result seen until the current iteration is saved. A tolerance parameter λ is used to control recovery in the sense that if the error gets greater than λ times the lowest error seen so far, the search is restarted at the best point found so far, but with attenuated learning and momentum rates. With a small probability, the algorithm can also start at a completely different point.

   Because of complexity reasons, we did not implement the recovery part.

This leads to the following equations:

$$\begin{aligned}
\Delta w_{ij}^{(k)} &= -\gamma_{ij}^{(k)} \partial E / \partial w_{ij}^{(k)} + \alpha_{ij}^{(k)} \Delta w_{ij}^{(k-1)} \\
\gamma_{ij}^{(k+1)} &= MIN(\gamma_{max}, \gamma_{ij}^{(k)} + \Delta\gamma_{ij}^{(k)}) \\
\alpha_{ij}^{(k+1)} &= MIN(\alpha_{max}, \alpha_{ij}^{(k)} + \Delta\alpha_{ij}^{(k)})
\end{aligned}$$

$$\Delta\gamma_{ij}^{(k)} = \begin{cases} \kappa_l e^{-\beta_l |\hat{S}_{ij}^{(k)}|} & \text{if } \hat{S}_{ij}^{(k-1)} \nabla E(W^{(k)}) > 0 \\ -\phi_l \gamma_{ij}^{(k)} & \text{if } \hat{S}_{ij}^{(k-1)} \nabla E(W^{(k)}) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

$$\Delta\alpha_{ij}^{(k)} = \begin{cases} \kappa_m e^{-\beta_m |\hat{S}_{ij}^{(k)}|} & \\ -\phi_m \alpha_{ij}^{(k)} & \text{see the conditions above} \\ 0 & \end{cases}$$

The parameters $\gamma_{max}, \alpha_{max}$ and $\kappa_l, \kappa_m, \phi_l, \phi_m, \beta_l, \beta_m$ have to be specified by the user; the expression $\hat{S}_{ij}^{(k)}$ is the same as in (15). For their experiments, Minai and Williams used the following values for the different parameters:

$$\begin{aligned}
\gamma_{max} &= 10.0 & \alpha_{max} &= 0.9 \\
\kappa_m &= 0.1 & & \\
\phi_l &= 0.3 & \phi_m &= 0.5 \\
\beta_l &= 20.0 & \beta_m &= 5.0
\end{aligned}$$

Experiments with different values for κ_l ($\kappa_l = 0.001$, $\kappa_l = 0.05$ and $\kappa_l = 1.0$) were made, and it turned out, that $\kappa_l \approx 0.05$ worked best.

3.3 Second-order algorithms

The idea behind these methods is that the error function $E(W)$ is approximated locally by a quadratic function, its truncated Taylor series

$$E(W^{(k)} + h) \approx E(W^{(k)}) + \nabla E(W^{(k)})^T h + \frac{1}{2} h^T \nabla^2 E(W^{(k)}) h,$$

where $\nabla^2 E(W)$ is the second derivative of $E(W)$, the *Hessian matrix* with

$$\nabla^2 E(W)_{kl} = \frac{\partial E^2}{\partial w_k w_l}.$$

To avoid the use of four indices, we refer here to the elements of W as w_k instead of w_{ij} as usual. The quadratic approximation above can now be minimized exactly. We just have to find the point $W^{(k+1)} = W^{(k)} + h$, where $\nabla E(W^{(k+1)}) = 0$ holds. We have

$$0 \;=\; \nabla E(W^{(k)} + h)^T \;=\; \nabla E(W^{(k)})^T + h^T \nabla^2 E(W^{(k)}),$$

and this leads to

$$h = -\nabla^2 E(W^{(k)})^{-1} \nabla E(W^{(k)}),$$

which means that $W^{(k+1)}$ is of the following form:

$$W^{(k+1)} = W^{(k)} - \nabla^2 E(W^{(k)})^{-1} \nabla E(W^{(k)}). \tag{17}$$

The minimum of the quadratic approximation so obtained is of course not the exact minimum of $E(W)$ itself, but we can iteratively compute $W^{(k)}$ until we are sufficiently close to the minimum of $E(W)$. Since a function behaves almost quadratically near its global minimum, the method converges faster, as we get nearer to a solution.
The iteration, which is defined by equation (17) is the pure form of *Newton's method*. It can be modified by introducing a parameter α. We get

$$W^{(k+1)} = W^{(k)} - \alpha^{(k)} \nabla^2 E(W^{(k)})^{-1} \nabla E(W^{(k)}),$$

were $\alpha^{(k)}$ is obtained in a line search to minimize E. As we get closer to the solution, $\alpha^{(k)}$ gets of course closer to 1.
However, Newton's method has several significant drawbacks, which make it problematic as a learning algorithm for neural networks. First, to obtain a good convergence-rate, we have to find a good initial value $W^{(0)}$, which is very hard for neural networks. The best initial values we can find are often random ones. Second, it is very expensive in both memory and time requirements. In each step, non-local information is needed in order to estimate the Hessian matrix $\nabla^2 E(W^{(k)})$ or its inverse.
To overcome these drawbacks, a second modification is made. We replace the inverse of the Hessian matrix at each step by a matrix $M^{(k)}$, which is supposed to be some (easy to compute) approximation of $\nabla^2 E(W^{(k)})^{-1}$. Since the Hessian matrix is positive definite at the minimum of $E(W)$, this must also hold for $M^{(k)}$. We get

$$W^{(k+1)} = W^{(k)} - \alpha^{(k)} M^{(k)} \nabla E(W^{(k)}), \tag{18}$$

whereby a again line search is required to estimate the $\alpha^{(k)}$ which minimizes $E(W^{(k+1)})$. Methods, which involve the steps defined by (18), including a line search in some cases to obtain $\alpha^{(k)}$,

are called *Quasi-Newton Methods.* Backpropagation as a gradient descent method belongs to this class of algorithms ($M^{(k)} = I$). This also holds for the adaptive step methods, since they also use $M^{(k)} = I$ and some sort of line search to estimate $\alpha^{(k)}$.
The objective now is to find matrices M_k which are a better estimation than I and easier to compute and store than $\nabla^2 E(W^{(k)})^{-1}$. One approach is the *the Davidon-Fletcher-Powell Method (DFP)* [Lue73, Wat87], which updates $M^{(k)}$ at each step as

$$M^{(k+1)} := M^{(k)} + \frac{p^{(k)}p^{(k)T}}{p^{(k)T}q^{(k)}} - \frac{M^{(k)}q^{(k)}q^{(k)T}M^{(k)}}{q^{(k)T}M^{(k)}q^{(k)}} \tag{19}$$

where $p^{(k)} := \alpha d^{(k)}$ minimizes $E(W^{(k)}+\alpha d^{(k)})$ along the search direction $d^{(k)} := -M^{(k)}\nabla E(W^{(k)})$, and $q^{(k)} := \nabla E(W^{(k+1)}) - \nabla E(W^{(k)})$. It can be shown by induction that all $M^{(k)}$ are symmetric positive definite, if $M^{(0)}$ is [Lue73]. It can also be shown that $p^{(i)T}\nabla^2 E(W)p^{(j)} = 0$ for $i \neq j$, which means that the $p^{(k)}$ are mutually $\nabla^2 E(W)$-conjugated. This makes the Davidon-Fletcher-Powell method (19) a cg-method.
Another more sophisticated approach is the **the Broydon-Fletcher-Goldfarb-Shanno Method (BFGS)** [Wat87]

$$M^{(k+1)} := M^{(k)} + (1 + \frac{q^{(k)T}M^{(k)}q^{(k)}}{p^{(k)T}q^{(k)}})\frac{p^{(k)}p^{(k)T}}{p^{(k)T}q^{(k)}} - \frac{p^{(k)T}q^{(k)}M^{(k)} + M^{(k)}q^{(k)}p^{(k)T}}{p^{(k)T}q^{(k)}} \tag{20}$$

where $p^{(k)}$ and $q^{(k)}$ have the same form as described above.
The drawback of these methods is, that although the computational effort is reduced by order $O(n)$, as compared to Newton's method, where n is the number of network's weights, both approaches require global information. For this reason we dropped them from our comparison. R. L. Watrous tested both methods (19) and (20) in [Wat87] using the XOR- and a 'moderate sized' multiplexer problem with 33 weights as benchmarks. He showed the BFGS method to be superior to the DFP method and both to be faster than standard backpropagation. Using these methods he also obtained some good results in speach recognition problems [WSW87].

3.3.1 The pseudo-Newton diagonal estimation algorithm

A quasi-Newton method that needs no global information has been proposed by S. Becker and Y. le Cun [BlC88]. Their idea was to neglect the off-diagonal elements of the Hessian matrix, that is

$$\nabla^2 E(W) \approx \begin{pmatrix} \ddots & & 0 \\ & \partial^2 E/\partial^2 w_{ij} & \\ 0 & & \ddots \end{pmatrix}.$$

This leads to the following learning rule:

$$\Delta w_{ij} = -\frac{1}{|\partial^2 E/\partial^2 w_{ij}| + \mu} \cdot \frac{\partial E}{\partial w_{ij}},$$

where $\mu > 0$ is a small constant introduced to avoid that the denominator gets too close to zero. The method described above has the drawback, that. because of the nonlinearity of the second derivatives, this algorithm can only be run in on-line mode.
Since the speed-up reported by Becker and le Cun is not significant enough, and on the other hand the computation of the diagonal elements of the Hessian matrix is long and complicated, we also dropped this method from our comparison.

3.3.2 Quickprop

Quickprop is an improvement to backpropagation, proposed by S. E. Fahlmann [Fah88], which is not a direct derivation of Newtons method, like the other second order methods described before. It is rather an adaptive step algorithm which assumes the error function to be quadratic and which performs a kind of Newton-iteration but in a single direction. It is a kind of 'Newton-line-search'. Quickprop is based on two assumptions:

1. The error function $E(W)$ is a parabola, whose arms open upward.
2. The change in the slope of the error curve, as seen by each weight, is not affected by all the other weights that are changing at the same time.

Now if the error function is quadratic in each direction and is not affected by other weights, the first derivative has to be linear.

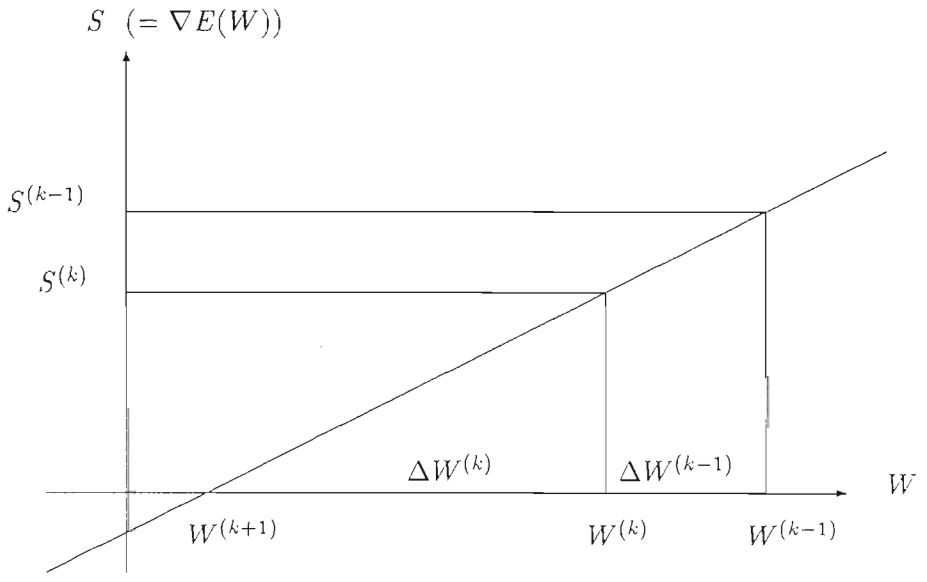


Figure 11: Visualisation of a Quickprop step

Under this assumptions and with $\Delta W^{(k-1)}$ already computed, we have to find $W^{(k+1)}$, where $\nabla E(W^{(k+1)}) = 0$ holds, which means, we have to determine $\Delta W^{(k)}$.
As Figure 11 suggests, this is easily done, because the equation

$$\frac{\Delta W^{(k)}}{S^{(k)}} = \frac{\Delta W^{(k-1)}}{S^{(k-1)} - S^{(k)}}$$

holds, with $S^{(k)} := \nabla E(W^{(k)})$, and thus we have for $\Delta W^{(k)}$

$$\Delta W^{(k)} = \left(\frac{S^{(k)}}{S^{(k-1)} - S^{(k)}}\right)\Delta W^{(k-1)}, \tag{21}$$

where the vector operations are performed componentwise.
Now we have to distinguish three different cases:

1. $S^{(k)}$ has the same sign as $S^{(k-1)}$ but is somewhat smaller. This means that the weights are again changed in the same direction.

2. $S^{(k)}$ and $S^{(k-1)}$ have opposite signs. This means that we have jumped over the minimum, $W^{(k+1)}$ will be somewhere between $W^{(k)}$ and $W^{(k-1)}$.

3. $S^{(k)}$ has the same sign and has the same size or is larger than $S^{(k-1)}$. Now we have to constrain the step size, because if we follow the updating formula blindly, it will lead to infinitly large steps. Fahlmann handles this situation by introducing a new parameter, the 'maximum growth factor μ', which means that we are not allowed to change the weights more than μ times $\Delta W^{(k-1)}$. Fahlmann claims, that $\mu \approx 1.75$ will do well.

Since the weight changes (21) always depend on the weight changes of the previous step, some bootstrap strategy is required to start the process, or restart it for those weights which have previously taken a step of size zero and are now facing nonzero gradient components, because of some changes elsewere in the network. In this case, Fahlmann uses an ordinary backpropagation step with a fixed learning constant γ and a momentum rate α.
Such a gradient descent term is always added to the weight change $\Delta W^{(k-1)}$ computed with (21), except in the case, where $\nabla E(W^{(k)})$ is nonzero and in the opposite direction of $\nabla E(W^{(k-1)})$. The updating (21) alone is enough then, otherwise we may overshoot the minimum again, which would lead to an oscillation of the learning algorithm.
Fahlmann observed that in some problems Quickprop causes some of the weights to grow very large, which may even produce floting point overflows during the learning phase. He handles this problem by adding a small weight decay (he chooses it to be $-0.0001 \times W_k$) to the gradient $\nabla E(W^{(k)})$ computed at each step. Since the goal is to take large steps (in the right direction) in weight space, batching is used to obtain the gradients $\nabla E(W^{(k)})$. Fahlmann also adds an offset to the derivative of the sigmoid (see 11) and uses constraint bipolar vectors (see section 3.1.3) to speed the algorithm up.
For some larger problems, he uses a technique proposed by D. C. Plaut, S. Nowlan and G. Hinton [PNH86], which chooses a different learning rate for every single neuron and its incoming weights, depending on the number of the neuron's incoming connections. This means that for neuron j the incoming weights are changed with a rate $\gamma_j = \gamma_{global}/(\#\text{Incom. weights of neuron } j)$. Fahlmann calls this technique, which we did not test, 'split epsilon'.

3.3.3 The extended Quickprop

M. Fombelida and J. Destiné merged the Extended Delta-Bar-Delta (see section 3.2.4) and the Quickprop Algorithm described above [FD92]. The idea was to introduce the adaptive learning rate of the Extended Delta-Bar-Delta Algorithm into Quickprop, which uses just an adaptive momentum rate (21). Extendended Quickprop has a structure more reminescent of Quickprop than of the extended Delta-Bar-Delta algorithm. We consider it to be a second order method too. The proposed transference of the adaptive learning rate into Quickprop can be described as follows, where the three distinguished cases correspond with those of the Quickprop Algorithm.

1. $sign(S_{ij}^{(k)}) = sign(S_{ij}^{(k)})$ and $|S_{ij}^{(k)}| < |S_{ij}^{(k)}|$.

$$\Delta W_{ij}^{(k)} := \begin{cases} -\gamma_{ij}^{(k)} S_{ij}^{(k)} + \alpha \Delta W_{ij}^{(k-1)} & if \quad v_{ij}^{(k)} > \alpha \Delta W_{ij}^{(k-1)} \\ -\gamma_{ij}^{(k)} S_{ij}^{(k)} + v_{ij}^{(k)} & otherwise \end{cases}$$

2. $sign(S_{ij}^{(k)}) = sign(S_{ij}^{(k)})$ and $|S_{ij}^{(k)}| \geq |S_{ij}^{(k)}|$.

$$\Delta W_{ij}^{(k)} := -\gamma_{ij}^{(k)} S_{ij}^{(k)} + \alpha \Delta W_{ij}^{(k-1)}$$

3. $sign(S_{ij}^{(k)}) \neq sign(S_{ij}^{(k)})$.

$$\Delta W_{ij}^{(k)} := \begin{cases} \alpha \Delta W_{ij}^{(k-1)} & if \;\; v_{ij}^{(k)} > \alpha \Delta W_{ij}^{(k-1)} \\ v_{ij}^{(k)} & otherwise \end{cases}$$

The learning rates are changed with the following rule

$$\Delta \gamma_{ij}^{(k)} := \begin{cases} \kappa e^{(-\beta |S_{ij}^{(k)}|)} & if \;\; \hat{S}_{ij}^{k-1} S_{ij}^{(k)} > 0 \\ -\phi \gamma_{ij}^{(k-1)} & if \;\; \hat{S}_{ij}^{k-1} S_{ij}^{(k)} < 0 \\ 0 & otherwise \end{cases}$$

Like in case of the Delta-Bar-Delta algorithm, we have for $S_{ij}^{(k)}$ and $\hat{S}_{ij}^{(k)}$

$$S_{ij}^{(k)} = \partial E / \partial w_{ij} \qquad \hat{S}_{ij}^{(k)} = (1-\theta) S_{ij}^{(k)} + \theta \hat{S}_{ij}^{(k-1)}$$

and $v_{ij}^{(k)}$ is the quadratic Quickprop step

$$v_{ij}^{(k)} = \left(\frac{S_{ij}^{(k)}}{S_{ij}^{(k-1)} - S_{ij}^{(k)}}\right) \Delta W_{ij}^{(k-1}.$$

The parameters α, κ, ϕ and θ have to be specified by the user. Since Fombelida and Destiné do not write anything about how they choose them, we choose them to be the same as in Quickprop and the Extended Delta-Bar-Delta Algorithm, that is $\kappa \approx 0.05$, $\phi \approx 0.3$ $\theta \approx 0.7$ and $\alpha = 0.9$. The learning rates γ_{ij} where again initialized to 0.1.

4 Orthonormalisation and Decorrelation of the Training-Set

This is in fact some kind of preconditioning, which means that the network is not trained with the original training set, but with a 'new' set, which is the result of some linear transformation being applied to the original one, to achieve orthonormalization, and thus decorrelation of the data. The positive effect of such a decorrelation can be explained by taking a closer look at the network architecture shown in Figure 1. The network can be divided in two parts:

1. **An associative memory:** From the input layer to the output of the hidden layer.
2. **A Linear regression network:** From the output of the hidden- to the input of the output layer.

The output layer may be neglected in this discussion, since it just performs a 'squashing' of its input, which is actually the important variable (see also section 3.1.5, item 3). It is easy to find the weights for the two separate parts, if both are trained independently. The difficulty of training a multilayer network is, that both parts have to be trained *simultaneously*, and of

course the targets of the associative memory (the output of the hidden layer), i.e the input of the regression part are unknown. Nothing remains constant during learning and experiments show, that *first* the weights of the first part (and thus the *internal representations*) are found, and after that the weights for the linear regression part are estimated.

A closer look at the theory of associative memory and the corresponding *hebbian learning* (see for example [HN91, HKP91, Roj93]) shows, that learning in associative memories becomes much easier (and also the capacity of the memory increases!), as the vectors that have to be learnt, are uncorrelated. A speedup in training multilayer perceptrons can also be expected, if the elements of the trainings set are uncorrelated or, even better, are orthonormal.

This may be visualized as a change of the error surface, which has to be climbed down. Since there is a relation (or *duality*, see [Roj93]) between input- and weight space, orthogonalization of the input data has also some kind of orthogonalizing effect on the error function in weight space. This means that small angles between different steps of the error function are increased, which has a 'rounding' effect on the error surface. Small narrow oval valleys become more symmetric now and much easier to descend.

4.1 Principal component ananlysis: Sanger's and Oja's Rule

Principal component analysis (PCA) is in fact some kind of data compression. The objective is to find m orthogonal vectors out of a set of n-dimensional input vectors, that account for as much as possible of the data variance [HKP91, Roj93]. The input data is projected into an m-dimensional subspace ($m << n$ is desirable), and each n-dimensional input vector is replaced by its representation with respect to the m principal components, i. e. an m-dimensional vector. Because of the possible reduction in dimensionality (and thus, reduction in the number of weights) this makes the data much easier to handle. If, because of the data variance, such a reduction can not be obtained, the computed principal components are at least mutually orthogonal which has the 'rounding' effect on the error surface described above.

These principal components are computed as follows. The first principal component is taken to be along the direction with the maximum variance, which is the n-dimensional vector c_1 that maximizes

$$\sum_{j=1}^{n} |c_{1j} x_{ij}|^2$$

for all input vectors x_i. The second principal component c_2 is constrained to lie in the subspace orthogonal to the subspace spanned by c_1. Within that subspace it is again in the direction with the maximum variance and is estimated by computing the principal component of the vectors $\hat{x}_i$, where $\hat{x}_i := (I_n - c_1 c_1^T) x_i$ is the projection of x_i onto the the subspace orthogonal to c_1.

The other principal components are computed in the same way. The m-th principal component c_m is obtained by computing again the principal component of the $\hat{x}_i$, where $\hat{x}_i := (\sum_{k=1}^{m-1}(I_n - c_k c_k^T)) x_i$, the projection onto the subspace orthogonal to $c_1, \ldots, c_{m-1}$.

Different strategies have been proposed to compute the first m principal components on a one-layer feed-forward neural network. These (unsupervised) networks, independently designed by Sanger and Oja [HKP91], use the following learning rules:

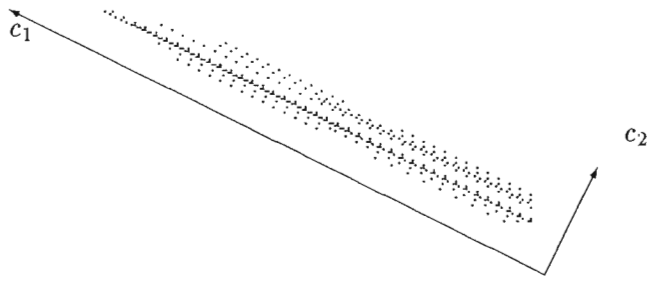


Figure 12: The first two principal components of a set of two-dimensional data

- **Sangers Rule**

$$\Delta w_{ij} = \beta c_i (x_j - \sum_{k=1}^{i} c_k w_{kj})$$

- **Oja's m-unit Rule**

$$\Delta w_{ij} = \beta c_i (x_j - \sum_{k=1}^{n} c_k w_{kj}),$$

which just differ in the upper limit of the summation. Unfortunately neither of these learning rules is local, so Sanger suggested a reformulation of his learning rule, which preserves locality.

$$\Delta w_{ij} = \beta c_i ((x_j - \sum_{k=1}^{i-1} c_k w_{kj}) - c_i w_{ij}) \tag{22}$$

The learning rate β has to approach zero slowly, as the number of iterations increase.
The corresponding network for this learning rule has the structure shown in Figure 14, where the neurons denoted with a '+' just compute the inner product of input- and weight vector, i. e. their activation function is just the identity.

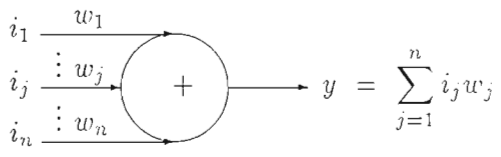


Figure 13: A linear associator and its computation

The idea is to compute the first principal component with a linear associator and subtract it from the input. The second linear associator computes the first principal component of this modified input, that is it computes the second principal component of the original input, and so on. The network is trained with Sanger's rule and after training is completed, backpropagation

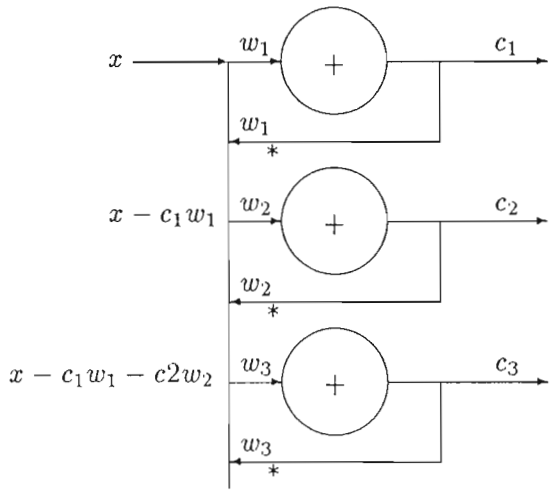


Figure 14: Sanger's network for the computation of the first 3 principal components

is performed replacing the original trainings input with the output $c_1 \ldots c_m$ of the network described above. The target vectors remain unchanged.
We trained the network using $40 \times (\#inputunits) \times (\#inputpatterns)$ training cycles. The learning rate was initially set to $\beta = 0.1$ and every $(\#inputunits) \times (\#inputpatterns)$ cycles reduced to $\beta := \beta * 0.75$.

4.2 Adaptive data-decorrelation

F. M. Silva and L. B. Almeida have proposed a different mrthod of data decorrelation and orthonormalisation [AS92, AS91a. AS91b]. They employ layers of linear associators after the input- and hidden layer, which have as much neurons as the input- rsp. hidden layer. These layers perform a linear transformation, such that the output is uncorrelated and normalized. These additional layers are trained with an unsupervised learning rule, either in batch or in on-line mode.
We first give a motivation of the learning rule. The output y_j of neuron j of a single layer of linear associators (see Figure 13 above), where the number of input and output neurons are equal, is

$$y_j = a_j^T x,$$

where x is the input vector and a_j is the weight vector of all incoming channels of output neuron j. The correlation of two given outputs y_j and y_k is

$$r_{ij} = E(y_i y_j) = a_i^T E(xx^T) a_j = a_i^T R_{xx} a_j, \tag{23}$$

where R_{xx} is the correlation matrix of the input data and $E(\cdot)$ is the expected value operator. We want to find now a_i, a_j, such that y_i and y_j become uncorrelated. This means that $r_{ij} = \delta_{ij}$, where δ_{ij} is the Kronecker-δ. Looking at (23) we notice, that this means that a_i and a_j have to be orthogonal in a space with metric R_{xx}. As suggested in Figure 15 below this is done by changing each vector a small amount β in a direction parallel to the other vector, weighted by the dot product between both.

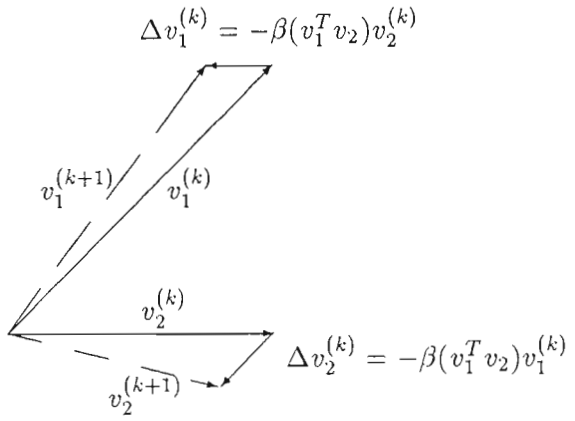


Figure 15: Iterative orthogonalization of the vectors v_1 and v_2

We just replace the dot product by the dot product with respect to R_{xx} to obtain for a_i

$$\begin{aligned} a_i^{(k+1)} &= a_i^{(k)} - \beta(a_i^{(k)T} R_{xx} a_j^{(k)}) a_j^{(k)} \\ &= a_i^{(k)} - \beta r_{ij} a_j^{(k)}, \end{aligned}$$

where β is a learning constant and k the iteration index.
The general case with N variables (for the proof see [AS91a] e.g.) is handled by the expression

$$a_i^{(k+1)} = a_i^{(k)} - \beta \sum_{j \neq i}^{N} r_{ij} a_j^{(k)}; \quad i = 1, \ldots N.$$

Since data *normalization* is also desired, this equation can be modified, so that a small contribution of the adapted vector itself is included, positive or negative, depending on whether r_{jj} is smaller or greater than 1.

$$\begin{aligned} a_i^{(k+1)} &= a_i^{(k)} - \beta \sum_{j \neq i}^{N} r_{ij} a_j^{(k)} + (1 - r_{ii}^{(k)}) a_i^{(k)} \quad i = 1, \ldots N. \\ &= (1+\beta) a_i^{(k)} - \beta \sum_{j=1}^{N} r_{ij} a_j^{(k)} \quad j = 1, \ldots N. \end{aligned} \tag{24}$$

In [AS91a] a proof is given that the algorithm above converges, if

- $A = (a_1, \ldots, a_N)$ is initially set to the identity matrix I_N,
- and $\beta < min(1/2\ ,\ \ 1/(3\lambda_{max}^x - 1)$,

where λ_{max}^x is the largest eigenvalue of R_{xx}.
Equation (24) is the batch version of the learning algorithm. The online version is derived by substituting r_{ij} by the product $y_i y_j$ and a time dependent learning parameter We obtain

$$a_{ij}^{(k+1)} = (1 + \beta^{(k)}) a_{ij}^{(k)} - \beta^{(k)} y_i^{(k)} \sum_{j=1}^{N} a_{ij}^{(k)} y_j^{(k)}; \quad i = 1, \ldots N.$$

The distributed implementation of this (on-line) algorithm can be obtained by rewriting the equation above as

$$a_{ij}^{(k+1)} = (1+\beta^{(k)})a_{ij}^{(k)} - \beta^{(k)} y_i^{(k)} z_j^{(k)}, \quad \text{where } z_j = \sum_{i=1}^{N} a_{ij}^{(k)} y_i^{(k)}; \quad j = 1, \ldots N. \tag{25}$$

The **learning phase of the on-line version** can now be divided into three steps

1. **Feedforward step.** Computation of $y_i = a_i^T x$.
2. **Backward step.** Computation of the z_j, which are stored in the units of the input layer.
3. **Weight update step.** As described in equation (25).

For the **batch version**, all the correlation parameters r_{jk} are known, and we have

$$r_{ij} = \frac{1}{P} \sum_{p=1}^{P} y_{pi}^{(k)} y_{pj}^{(k)},$$

where p denotes the training pattern and P the size of the training set. This algorithm can also be implemented using the same architecture, since (24) can be rewritten as

$$a_{ij}^{(k+1)} = (1+\beta^{(k)})a_{ij}^{(k)} - \frac{\beta^{(k)}}{P} \sum_{p=1}^{P} y_i^{p(k)} z_j^{p(k)},$$

where z_i^p denotes z_i, computed for the training pattern p.
The learning rate was initially set to $\beta = 0.01$ and every P, but at least 50 and at most 100 cycles, it is reduced using $\beta = \beta * 0.33$, with a lower boundary $\beta_{min} = 0.0001$.

5 Testing conditions

In this section, we describe some factors that have a great impact on the required learning time, and what we did about them.

5.1 Stopping Criteria

One difficulty is how to take the decision, when to stop the learning algorithm and conclude that the learning phase is complete. The fastest learning algorithms degrade, if they have to run several times longer as necessary, because of a too stringent stopping condition.
Some of the commonly used stopping criteria are the following ones:

- **Small composite error:** This simply means, that learning is assumed to be finished, whenever $E(W) < \epsilon$ for some small ϵ (usually around 0.01) holds.
- **Small individual error:** Each output is required to be very close to the corresponding component of the target, which mathematically means, that $\|o^{(2)} - t\|_\infty := \max_j |o_j^{(2)} - t_j| < \epsilon$ has to hold.
- **Sharp threshold:** Every component of the output is regarded as one, if $o_j^{(2)} > 0.5$ and regarded as zero, if $o_j^{(2)} < 0.5$. The output is indefinite for $o_j^{(2)} = 0.5$.

- **Threshold with margin:** A compromise between the sharp threshold criterium and the small individual error criterium. The output components $o_j^{(2)}$ are regarded as one for $o_j^{(2)} > (1-\sigma)$ and as zero for $o_j^{(2)} < \sigma$, and are indefinite otherwise. The choice of $0.5 < \sigma < 1$ depends on the problem to be learned.

Some other criterias are listed in [Fah88].
We think, like Fahlmann, that the first criterium (small composite error) is the worst choice, although it is used by many researchers. On one hand we do not want some large error in one output component to be traded off against very small errors in the others, since we want *all* outputs to have small individual errors. On the other hand, the criterium is too restrictive for many applications, because we do not want the output to be *exactly* zero or one, for example. We just want to be able to tell what the output is supposed to be, for example using the sharp threshold criterium. In some benchmarks we had all the output bits classified right, although $E(W)$ was not smaller than 0.5!
To chose the right stopping criterium, one has to pose two questions about the problem to be learned:

1. What kind of output do we want, real or binary?
2. Is the problem completely represented by the training set, or is some interpolation required by the network?

If the target vector consists of real numbers, there is only one stopping criterium that makes sense, the small individual error criterium, no matter if the network has to do interpolation or not. We can not choose something like a threshold with margin criterium (margin around *what*?) and the small composite error criterium has the drawbacks discussed above.
If we have binary outputs on the other hand, the question of interpolation has an impact on the choice of the stopping criterium.
If there is *no interpolation*, which means that the network works as some kind of associative memory, the sharp threshold criterium is sufficient. Because the network 'knows' *all* the possible patterns, after the training phase has been completed, we can definitely tell, when an output is supposed to be one or zero.
If there is *interpolation*, we want the network to produce for an unlearned pattern $\hat{i}$ close to a known pattern i an output $\hat{o}$ close to the output o belonging to i. So during the training phase, we have to force the network to produce an output that is sufficiently close to the target vectors. If the sharp threshold criterium is too weak, the threshold with margin criterium will do. The margin σ is has to be chosen empirically to optimize the network.
We choose the stopping criteria discussed above depending on the problems to be learned, but we always explicitly mention the criterium used for the special tests.

5.2 Restarting Algorithms

It may frequently happen, that the learning algorithm gets stuck in some local minimum or in flat spots of the error function. In this cases, we allow the algorithm to start again with a different set of random weights. The total learning time then includes of course the time for the unsuccessful trials.
The problem is how to decide early enough when an algorithm has got stuck, so that it is restarted as soon as possible. On the other hand we want to be absolutely sure, that the

algorithm reached a local minimum. We do not want to be too hasty with the restart; maybe the algorithm can still recover from an unfavorable position.
Some researchers have proposed a restart if the algorithm has not found a solution within a certain number of iterations [RHW86, Fah88]. This of course only makes sense, if one knows how many iterations are usually required for a certain problem. If this is unknown, or if we want to 'tune' certain parameters of an algorithm to investigate their impact on the learning time, we need some other criterium to decide, when an algorithm has got lost.
We propose a rather simple test, that only requires the observation of the error function $E(W)$. This test requires no extra computation and of course no global information. We monitor the error function $E(W)$ to see how much it decreases during a certain number of iterations. If the decrement is too small, either because of a large flat plateau of the error function, or because of a local minimum, we restart the algorithm. We choose the number of iterations that we wait, before the decrement is estimated, to depend on the number of the patterns to be learnt. In fact we choose it to be twice the number of training patterns, but at least 40. If the error has not decreased at least 10^{-8} after this period, the algorithm will be restarted.

6 Benchmarks for Neural Networks

The problem of selecting adequate benchmarks to measure the convergence speed of a learning algorithm is a rather difficult one. A learning algorithm, that performes very fast on a certain problem may completely fail in another. We need a set of carefully choosen benchmarks, hopefully representative enough of 'real world' applications, to investigate what a certain algorithm is able to do.
The benchmark most often used is the XOR problem, because this is the only boolean function of two input variables (together with its negation), that can not be learned by a single perceptron. We ourselves did not use it as a benchmark, not even for 'nostalgic' reasons, because it is too small and not very relevant.
Tasks that can be solved by backpropagation networks can (roughly) be divided in two classes. On one hand, we have problems, which require a very sharp distinction between different input vectors. An example are extensions of the XOR problem, the so called *parity problems*. These require a network output of 1, if the input vector has an odd number of components equal to 1, and an output of 0, if this number is even. Two different input vectors with a hamming distance (the number of different bits) of 1, which means that they are as close together as possible, should produce a completely different output.
On the other hand we have problems that require some kind of interpolation or generalization, that is input vectors with a small hamming distance produce also similar outputs. Such problems are for example the *classification tasks* or the approximation of a given function by the network. Although both classes must be considered, we believe that problems belonging to the second class are closer to 'real world' tasks.

6.1 The Benchmarks used in this Paper

We selected, as a representative of the first class of benchmarks, a 4 bit parity problem.
We also made some runtime comparisons with *decoder/encoder tasks* of different sizes. The networks solving this kind of problems have (usually) n input units, $m < n$ hidden units and again n output units. The input consists of an n-dimensional vector with only one component

equal to one and the other components equal to zero. The task is to reproduce the input, after going through the 'bottleneck' of the hidden units. If $m = log_2 n$, the encoder is called *tight*, if $m > log_2 n$, it is called *loose*. These networks actually perform some kind of data compression and the corresponding decompression.
Here is a complete list of the benchmarks we choose to evaluate the performance of the different backpropagation variations.

- A 4-bit parity problem.
- Encoder/Decoder tasks of the sizes 10-5-10 and 16-4-16.
- A classification task, where the 16×16-square was divided into different clusters (Figure 16).
- A classification task, where from five input-components two were chosen by random out of the $(0, 1)$ intervall. The other three components were set to be linear combinations of the random ones. The target for each vector out of this data set (100 vectors) consisted of three reals, the maximum, minimum and average of the input vector.

  This task is of course not too exeiting, the aim was to have a benchmark with highly correlated input data.

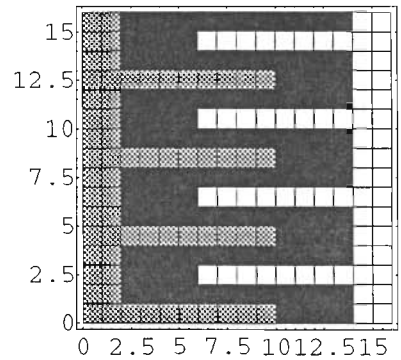


Figure 16: Clustering of the 16×16-square used as a learning benchmark.

7 Simulation Results

This section gives the runtime comparison of all the different variations described above. We first describe the network architectures and parameters that were used to solve the problems, then show the results and an interpretation of our experiments.

7.1 Architecture of the networks used

Table 1 shows the sizes of the networks, that were used to solve the benchmarks described above. A dimension referred to as 'X-Y-Z' means, that the network has X input-, Y hidden- and Z output units.
We also used an upper limit for the number of iterations. If no solution was reached within a certain number of training cycles, the algorithm was declared to have failed to converge.
The third number is the number of trials for each benchmark.

Benchmark	Network	Max.It.	Trials
10-5-10 Encoder	10-5-10	1.000	10
16-4-16 Encoder	16-4-16	1.000	10
4 bit Parity	4-4-1	2.000	10
Cluster Recognition	8-10-3	2.000	10
Rec. of correlated cluster	5-10-3	1.000	10

Table 1: Sizes of the networks and parameters used

The training data for the parity and encoder problems was binary, coded as described in section 6.1. For the cluster recognition problem we used binary input and output data. One output unit should identify each cluster. This unit was set to 1 if the input belonged to the corresponding cluster and set to 0 otherwise.
We choose as stopping criterium for the learning phase threshold with margin. For the parity and encoder problems, which are totaly represented by the training data, we choose a margin of 0.4, which is almost the sharp threshold criterium. For the cluster recognition problems, where the network has to interpolate, since we only used 190 out of the posible 256 points to train the network, we used a margin of 0.3. For the correlated cluster recognition problems we used a margin of 0.1, since the output consists of reals.
The network's weights were initialized with small real random numbers $r \in (-0.7, 0.7)$.

7.2 Runtime comparison

A runtime comparison with all the algorithms described in section 3.2 and 3.3 was made. We also tested the behavior of all algorithms when the following modifications, described in 3.1 were made

- Binary / Bipolar - Vectors.
- Adding an offset to the sigmoid's derivative.
- The use of decorrelation Algorithms (either PCA or Adaptive Decorrelation).
- The use of the modified error function (12).

This meant 24 possible combinations, which were all tested with the different benchmarks described above. The only exception was made for the encoder-problems. Since their input data consists of the unitary vectors of the input space, which are allready orthonormal, we did not perform PCA analysis on the input data. Our experiments show that performing PCA in this cases rather slows the convergence process down.
The possible variations are referred to as 'Gradient-Reuse-05' for example, were the extensions are described in the following table:

	Vec. Mode	Decor.	Offset	Err.Func		Vec. Mode	Decor.	Offset	Err.Func
01	Binary	None	0.0	Standard	02	Bipolar	None	0.0	Standard
03	Binary	None	0.1	Standard	04	Bipolar	None	0.1	Standard
05	Binary	PCA	0.0	Standard	06	Bipolar	PCA	0.0	Standard
07	Binary	PCA	0.1	Standard	08	Bipolar	PCA	0.1	Standard
09	Binary	Ad.Dec.	0.0	Standard	10	Bipolar	Ad.Dec.	0.0	Standard
11	Binary	Ad.Dec.	0.1	Standard	12	Bipolar	Ad.Dec.	0.1	Standard
13	Binary	None	0.0	Modified	14	Bipolar	None	0.0	Modified
15	Binary	None	0.1	Modified	16	Bipolar	None	0.1	Modified
17	Binary	PCA	0.0	Modified	18	Bipolar	PCA	0.0	Modified
19	Binary	PCA	0.1	Modified	20	Bipolar	PCA	0.1	Modified
21	Binary	Ad.Dec.	0.0	Modified	22	Bipolar	Ad.Dec.	0.0	Modified
23	Binary	Ad.Dec.	0.1	Modified	24	Bipolar	Ad.Dec.	0.1	Modified

Table 2: Variations of the algorithms tested.

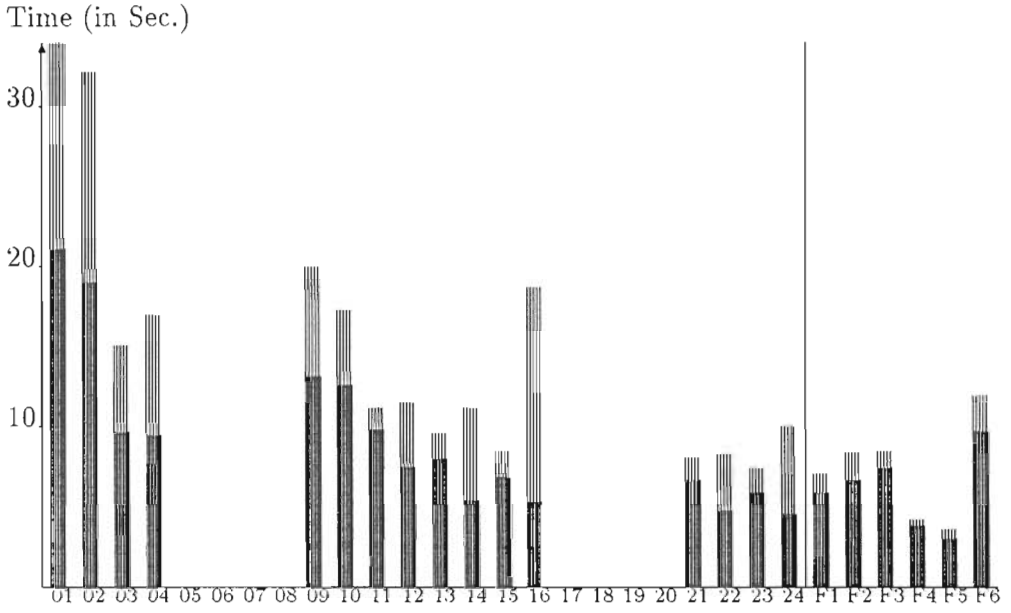
Testing eight algorithms with possible 24 variations on five different benchmarks means a huge mass of data. Showing all the results we obtained in tables would mean a 'table-overkill'. To avoid it, we decided to present the data in the following way:
For each benchmark, we report the results for all 24 (rsp. 16) variations on standard backpropagation in batch mode under the use of a momentum term. We report the mean training time as well as the fastest result. This is done to have the possibility to show and compare the speedups obtained.
Then we report the best result for each of the other algorithms. The rest of the results are presented in tables in the appendix, were we only report the results for those algorithms, that converged in at least 50% of the trials.
The diagrams show the average runtime (grey bars) as well as the fastest result obtained (black bars). The numbers 01, ..., 24 stand for the standard backpropagation variations (see Table 2 above). F1, ..., F6 stands for

F1	Fastest Result for	gradient reuse (GR)
F2	Fastest Result for	delta bar delta (DBD)
F3	Fastest Result for	extended delta-bar-delta (EDBD)
F4	Fastest Result for	dynamic adaption (DA)
F5	Fastest Result for	Quickprop (QP)
F6	Fastest Result for	extended Quickprop (EQP)

The corresponding constellation is reported below, like 'QP 22'.
If anything worth reporting occurred, that the figures and tables would not reveal (for example that some algorithm had difficulties, if trained with bipolar vectors or something like that), we describe the problem in the text.

7.3 Results for the 10-5-10 Encoder

The network for the 10-5-10 Encoder is relativly small, and the problem itself is rather easy to learn. So all the algorithms peformed relativly well, as Figure 18 shows.



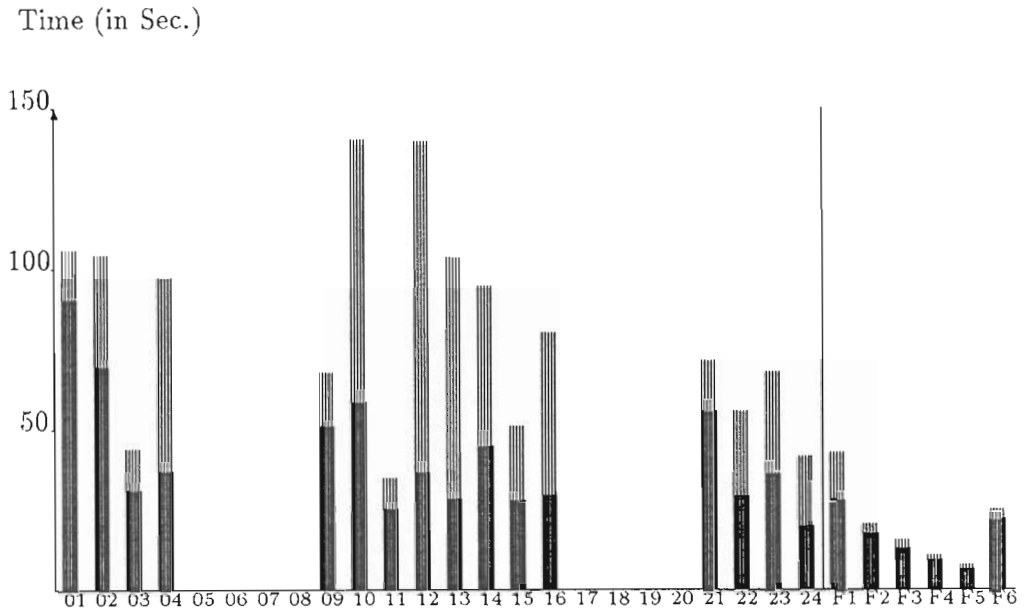
F1:	GR	03	F2:	DBD	23	F3:	EDBD	03
F4:	DA	15	F5:	QP	21	F6:	EQP	11

Figure 17: Results for the 10-5-10 Encoder problem.

It should be recognized that even simple modifications of the standard algorithm, like using bipolar vectors and/or adding an offset to the sigmoid's derivative, lead to significant speedups. Quickprop and the dynamic adaption algorithm learned the problem really fast, but this does not hold for the extended Quickprop.

7.4 Results for the 16-4-16 Encoder

The 16-4-16 Encoder is somewhat harder to learn than the 10-5-10 Encoder, but Figure 19 shows, that significant speedups can be obtained by using simple modifications of the standard algorithm.
The gradient reuse algorithm had great problems with this benchmark. The modified error function provided (in general) no speedup, it rather caused the algorithm to diverge. While Quickprop and the dynamic adaption algorithm still learned the problem fastest, extended Quickprop still was not too fast.



F1:	GR	03	F2:	DBD	13	F3:	EDBD	03
F4:	DA	03	F5:	QP	03	F6:	EQP	03

Figure 18: Results for the 16-4-16 Encoder problem.

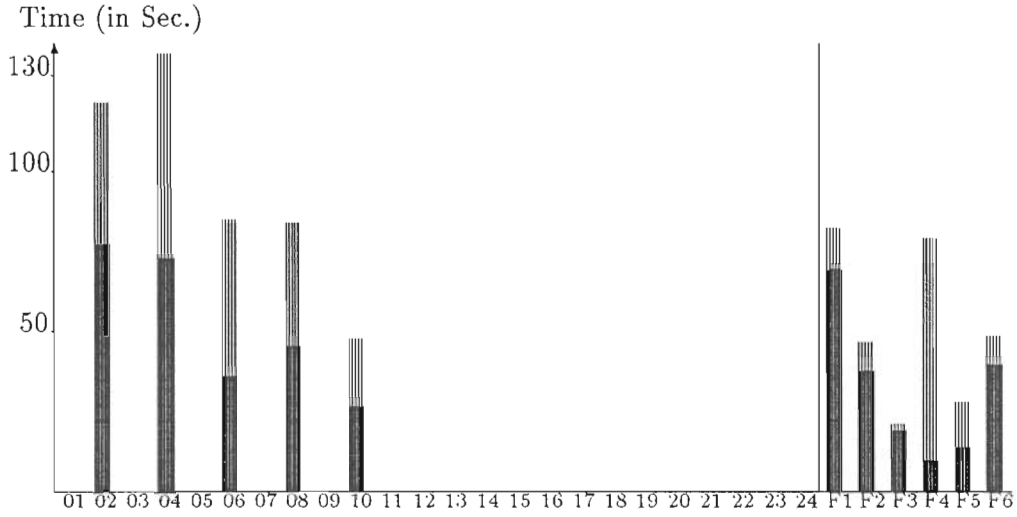
Almost all algorithms performed best with the same combination of binary vectors and the addition of an offset to the sigmoid's derivative.
It seems that the Encoder tasks are better learned with binary vectors, since the input is already orthogonal and decorrelated, which does not hold for the bipolar vectors anymore.

7.5 Results for the 4 Bit parity problem

This case is a lot more difficult to solve than the Encoder problems and in fact only five standard backpropagation variations converged at all. It is significant, that all five variations used bipolar instead of binary vectors to train the network.
The gradient reuse algorithm almost completely failed to converge (except for constellation 10) and also the dynamic adaption algorithm and Quickprop did not do too well. It is interesting that Quickprop reduced the error very fast at the beginning, but when the error was somewhat below 0.9, there was almost no reduction anymore. This could be solved by allowing bigger step sizes which may, on the other hand, lead to oscillations for other problems. Both delta-bar-delta algorithms, plain and extended, performed very well under almost all constellations.
It seems that what worked best for the 4 bit parity problem is the inclusion of the decorrelation layers, either with binary vectors or with bipolar vectors using the modified error function. PCA preconditioning provided almost no speedup, which is not too surprising, since there is no correlation between the input data.

7.6 Results for the cluster recognition problem

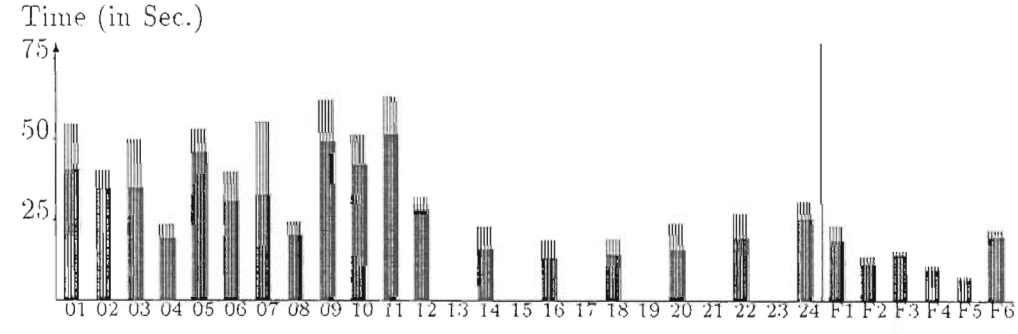
The cluster recognition problem now is the first benchmark, where the network has to interpolate between the training data, which it did well.



F1:	GR	10	F2:	DBD	10	F3:	EDBD	22
F4:	DA	22	F5:	QP	06	F6:	EQP	10

Figure 19: Results for the 4 bit parity problem.

It turned out, that for this problem the network performed sifgnificantly better using bipolar vectors, as Figure 21 shows. All the other standard modifications also speed the algorithm up, but none as bipolar vectors. The PCA preconditioning provides a small speedup for the pure training phase of the backpropagation network, but if one includes the training time for the PCA network, which is about 17 seconds, the speedup is gone.
This is almost the fact with the decorrelation layers. Although they reduce the iterations which are used to train the network (BP 02: 135 Iterations; BP 10: 87 Iterations), passing the informations through these layers takes more time than is saved by the iteration reduction. If the problem to learn gets more difficult, the decorrelation layers will pay.



F1:	GR	09	F2:	DBD	06	F3:	EDBD	06
F4:	DA	18	F5:	QP	14	F6:	EQP	06

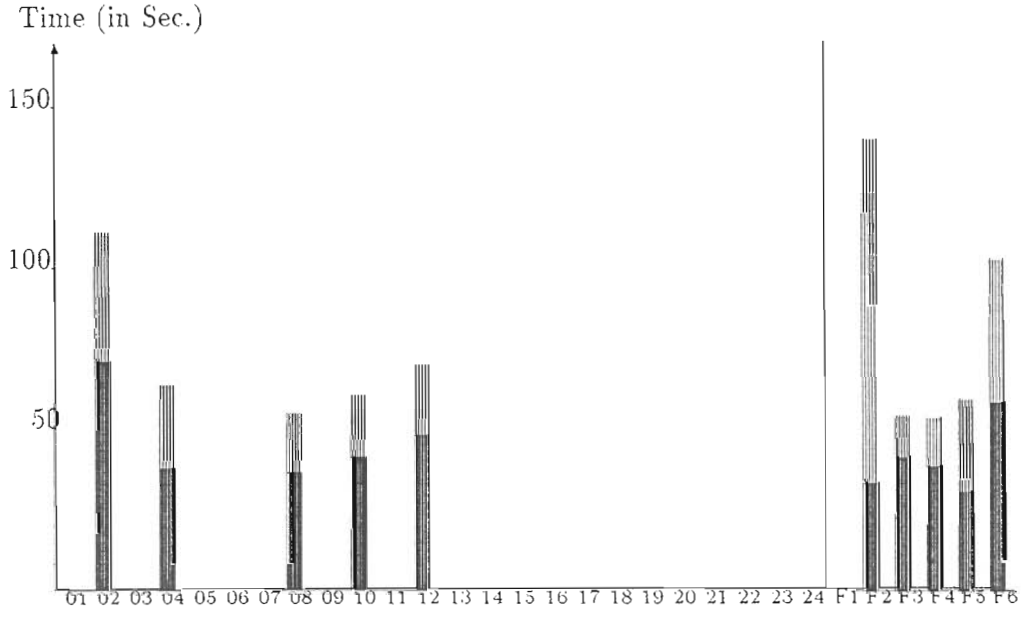
Figure 20: Results for the cluster recognition problem.

As for the parity problem, the gradient reuse algorithm performed relatively bad. in fact it was

often slower than the corresponding standard algorithm.
The DBD and EDBD algorithms performed again relatively well, although they had even more problems when dealing with binary vectors than the corresponding standard algorithm. The fastest results were again obtained with Quickprop and the dynamic adaption algorithm. Extended Quickprop again performed relativly poor, which means that if it converges at all, it does this rather slow.

7.7 Results for the correlated cluster recognition problem

This task was the hardest of them all, and as for the first cluster recognition, the algorithms performed significantly better under the use of bipolar vectors, as Figure 22 below shows. In fact, standard backpropagation only converges when trained with bipolar vectors.



F1:	F2: DBD 11	F3: EDBD 10
F4: DA 10	F5: QP 08	F6: EQP 22

Figure 21: Results for the correlated cluster recognition problem.

The table also shows, that all algorithms learn the task a lot faster, if decorrelation algorithms are used. The fastest results were all obtained if either PCA preconditioning was used (as for standard backpropagation and both Quickprop algorithms) or adaptive decorrelation layers were included, where the adaptive decorrelation algorithm seem to do a little better than PCA. This may have its reason in the fact, that PCA (where the input dimensionality is reduced to three) provides only a decorrelation of the input data, where the adaptive decorrelation algorithm decorrelates also the input of the hidden layer.
The gradient reuse algorithm fails completely to converge for all variations and the delta-bar-delta algorithm performed relatively bad, as well as the extended Quickprop. The extended delta-bar-delta algorithm, the dynamic adaption algorithm and Quickprop did very well. The

extended Quickprop algorithm converged in almost all cases, but the dynamic adaption algorithm and Quickprop were usually faster, where the dynamic adaption algorithm usually performed best.
The modified error function seems not to have a positive effect on the convergence speed, almost all algorithms converged faster without it.

7.8 Concluding remarks on the learning procedures

- **Standard variations**

  1. **Bipolar vectors** seem to have a major influence on the convergence speed of all learning procedures tested. Almost all benchmarks (except for the encoder problems, since their input is already orthonormal), were learned faster, sometimes significantly, when the network was trained with bipolar vectors, no matter which algorithm was used.
  2. **Adding an offset to the sigmoid's derivative** speeds the algorithms up in most cases. Exceptions are those problems, which require a fine tuning of the weights, like for example the four (or more) bit parity problems or the approximation of continuos functions, which we have not tested here. In these cases, in which the error function seems to have rather steep slopes anyway, the offset can cause the weight updating algorithms to perform rather wild jumps.
  3. **Decorrelation Algorithms** also seem to have a major influence on the convergence of the learning procedures. Uncorrelated data is in fact much easier to learn than correlated data. The speedup obtained with those decorrelation algorithms may even be bigger, if for example a network has to learn large sets of highly correlated experimental data. This can easily happen in real world applications.
  4. **The use of the modified error function.** Although it sounds like a good idea, the modified error function only seems to provide speedups for smaller, less complex tasks, such as the 10-5-10 encoder or the recognition of the shading of the 16×16-square. For more complex tasks, the modified error function rathe slows down the convergence, or even causes divergence of the algorithm. This may be ovecome, if the input error is not approximated (since the sigmoid is approximated by a straight line), but really computed. Then more precise error corrections without flat spots in the output layer can be expected.

- **Weight updating strategies**

  1. **The gradient reuse algorithm** was only faster than the standard algorithm for small problems, like the encoder problems or the recognition of the shading of the 16×16 square. For more complex problems it was either slower than standard backpropagation (like for the 4-bit parity problem) or did not converge at all (like for the correlated cluster problem).
  2. **The delta-bar-delta algorithm** only performed well for the 4-bit parity problem. It seems, that a fine tuning of the parameters κ, ϕ and ψ, on which the weight-updating depends, is required. This requires, on the other hand, quite a few tests to tune them, which again takes a lot of time.

3. **The extended delta-bar-delta algorithm** has similar problems as the delta bar delta algorithm, although it performes better in general, and significantly better for the recognition of the correlated cluster. But here as well, there are a lot of parameters which have to be chosen careful for each benchmark, which requires a lot of time and knowledge about the error surface shape.
4. **The dynamic adaption algorithm.** Although the algorithm performs very well for all benchmarks, it seems that the normalization of the gradient, which the original algorithm requires, has a bigger influence on the convergence of the algorithm than it was reported by Salomon. Theoretically it makes no difference, but in practice, if a large learning parameter and a steep descent come together, the algorithm may overshoot any minimum and get stuck in very flat spots far away from any solution. This was observed frequently, mainly for more complex problems. Since normalization of the gradient can not be done without using global information, another way may be figured out to constrain the stepsizes and stop the algorithm from jumping too wild. Besides this problem, the dynamic adaption has the big advantage, that there is no parameter, which needs to be tuned.
5. **Quickprop** also performed very well for all benchmarks. The algorithm has just one important parameter involved, the 'maximum growth factor μ', which almost needs no tuning. Experiments show, that it is rather to be choosen too small than too big, values of about 1.3 to 1.7 will do well for any problem.
6. **Extended Quickprop** was almost a complete failiure. It was often slower than the standard algorithm (or at least not significantly faster) and always slower than any other algorithm, except for the gradient reuse algorithm. This may again have the reason in the fact that there are quite a lot of parameters involved, which need to be tuned. In our opinion, an algorithm which has many parameters of great influence, which have to be tuned finely and which are different for each problem, is not very good anyway.

8 Future work

Having implemented and tested a great number of learning algorithms for multilayer neural networks, we are planning to design some kind of 'hybrid learning algorithm' by combining some the algorithms above. The algorithm should be able to switch between several optimizing strategies, depending on the shape of the error surface.
It is also planned to implement some of the algorithms, as well as the hybrid algorithm, on a massively parallel machine, the CNAPS computer of Adaptive Solutions. Since this machine is several times faster than the workstation on which the algorithms have been implemented so far (speedups of 500 - 1000 can be expected), much more complex tasks could be chosen as benchmarks. This will yield a better comparison of algorithms and will make it possible to solve real world tasks.

9 Appendix (Tables for the runtime comparison)

9.1 Results for the 10-5-10 Encoder

Benchmark: 10-5-10 Encoder														
	B-BP		GR		DBD		EDBD		DA		QP		EQP	
	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR
01	34.0	21.1	11.6	9.9	17.7	14.6	10.5	8.5	5.7	5.3	4.3	3.6	18.7	14.3
02	32.7	19.6	12.8	6.9	19.7	15.3	18.3	12.8	6.9	5.2	11.6	5.8	20.4	17.0
03	15.1	13.0	7.0	5.9	11.6	9.7	8.5	7.4	4.8	4.5	3.9	3.3	13.2	11.2
04	17.0	9.5	12.8	5.2	15.1	11.4	16.4	10.4	5.7	4.3	9.8	6.2	15.4	11.9
05	—	—	—	—	—	—	—	—	—	—	—	—	—	—
06	—	—	—	—	—	—	—	—	—	—	—	—	—	—
07	—	—	—	—	—	—	—	—	—	—	—	—	—	—
08	—	—	—	—	—	—	—	—	—	—	—	—	—	—
09	20.1	13.1	10.4	15.1	15.6	11.9	10.6	8.0	6.0	5.6	8.7	3.6	16.3	12.1
10	17.3	12.6	11.1	6.5	15.9	12.3	14.9	10.4	6.7	5.3	6.6	5.2	17.2	14.2
11	11.2	9.8	7.5	6.2	10.8	8.8	8.5	7.1	5.1	4.8	4.7	3.4	11.9	9.9
12	11.5	7.5	10.7	5.3	12.7	10.3	13.5	10.1	5.9	4.4	9.8	5.4	13.2	11.1
13	9.6	8.0	17.0	6.7	10.0	8.2	8.9	7.8	4.3	3.8	3.6	2.9	14.8	12.6
14	11.2	5.4	18.4	5.9	10.7	7.7	11.8	9.4	8.7	3.7	6.2	5.0	14.5	11.9
15	8.5	6.8	15.8	7.1	9.4	7.6	9.0	7.4	4.2	3.8	3.8	2.9	15.3	12.0
16	18.9	5.3	20.6	5.7	12.4	9.2	14.2	9.7	5.7	4.0	12.7	6.1	17.4	11.5
17	—	—	—	—	—	—	—	—	—	—	—	—	—	—
18	—	—	—	—	—	—	—	—	—	—	—	—	—	—
19	—	—	—	—	—	—	—	—	—	—	—	—	—	—
20	—	—	—	—	—	—	—	—	—	—	—	—	—	—
21	8.1	6.7	23.7	7.5	9.0	7.5	9.5	7.8	4.6	4.1	3.6	3.0	12.3	10.2
22	8.3	5.0	21.4	5.5	8.9	7.4	11.7	9.3	13.0	4.0	7.9	3.9	12.3	10.4
23	7.4	5.9	35.8	19.2	8.4	6.7	8.7	7.1	4.5	4.0	4.3	2.9	13.0	9.8
24	10.0	4.5	24.9	6.3	9.8	7.5	12.1	8.6	6.4	4.2	11.8	5.6	15.6	10.6

9.2 Results for the 16-4-16 Encoder

Benchmark: 16-4-16 Encoder														
	B-BP		GR		DBD		EDBD		DA		QP		EQP	
	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR
01	105.9	90.4	103.1	57.4	36.2	32.0	22.3	16.2	38.4	8.8	11.3	6.3	40.0	34.7
02	104.2	96.6	207.8	100.0	76.7	48.8	54.0	39.3	73.3	18.0	119.5	16.2	67.1	88.5
03	43.8	31.0	42.6	28.3	21.5	19.2	15.3	12.7	10.6	9.1	7.8	6.3	25.0	22.0
04	97.0	37.1	—	—	—	—	—	—	49.1	19.3	—	—	—	—
05	—	—	—	—	—	—	—	—	—	—	—	—	—	—
06	—	—	—	—	—	—	—	—	—	—	—	—	—	—
07	—	—	—	—	—	—	—	—	—	—	—	—	—	—
08	—	—	—	—	—	—	—	—	—	—	—	—	—	—
09	67.8	51.1	72.5	46.3	33.2	29.1	23.9	20.1	32.8	11.0	—	—	37.8	33.4
10	53.2	35.0	140.2	48.5	37.6	33.0	35.8	27.7	24.0	15.1	90.3	15.8	40.4	35.1
11	34.9	25.2	56.2	38.7	24.8	18.3	17.8	14.2	16.1	10.0	—	—	29.1	22.1
12	138.7	36.6	—	—	100.5	52.2	72.2	31.7	44.9	20.3	163.2	35.7	66.5	33.9
13	103.4	28.4	109.2	57.9	20.4	17.4	21.6	15.3	22.1	7.2	6.8	5.7	43.3	29.3
14	94.6	44.6	183.0	100.0	55.4	35.9	62.1	32.9	39.8	10.8	—	—	79.6	53.5
15	51.0	27.8	98.4	74.1	21.1	17.8	21.0	14.6	14.3	9.6	8.6	7.0	38.5	27.3
16	80.1	29.6	—	—	—	—	—	—	169.6	31.4	121.9	71.0	—	—
17	—	—	—	—	—	—	—	—	—	—	—	—	—	—
18	—	—	—	—	—	—	—	—	—	—	—	—	—	—
19	—	—	—	—	—	—	—	—	—	—	—	—	—	—
20	—	—	—	—	—	—	—	—	—	—	—	—	—	—
21	71.3	27.6	164.5	96.0	22.8	19.2	69.5	18.3	52.2	7.7	—	—	38.4	29.0
22	55.8	29.2	—	—	45.3	23.4	54.0	25.0	—	—	49.9	22.0	61.8	36.3
23	68.0	36.4	—	—	34.4	22.1	23.2	18.0	132.5	16.1	73.5	43.8	44.9	29.3
24	41.5	19.7	—	—	—	—	183.6	45.6	—	—	146.5	39.6	190.1	100.0

9.3 Results for the 4-bit Parity Problem

Benchmark: 4-bit Parity Problem														
	B-BP		GR		DBD		EDBD		DA		QP		EQP	
	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR
01	—	—	—	—	—	—	84.1	32.7	—	—	—	—	—	—
02	121.6	77.3	—	—	77.7	71.2	44.1	29.6	142.1	18.7	—	—	93.9	71.7
03	—	—	—	—	194.9	100.0	—	—	—	—	—	—	—	—
04	136.8	72.9	—	—	100.5	58.3	39.9	29.4	202.7	96.7	98.4	17.4	87.2	65.1
05	—	—	—	—	—	—	44.0	31.3	—	—	—	—	—	—
06	85.0	36.7	—	—	70.4	45.4	36.2	25.4	95.4	22.8	28.0	13.9	66.1	46.9
07	—	—	—	—	134.8	58.8	63.2	33.5	—	—	—	—	152.3	52.6
08	84.1	45.5	—	—	73.9	74.4	35.0	21.4	—	—	101.4	17.4	73.0	45.9
09	—	—	—	—	—	—	—	—	—	—	—	—	—	—
10	47.9	26.4	82.5	69.3	46.6	47.6	30.0	21.2	—	—	—	—	48.7	39.7
11	—	—	—	—	107.9	40.5	42.2	20.9	106.1	26.1	120.6	57.7	127.2	56.2
12	99.1	58.2	—	—	84.6	40.3	26.5	20.8	134.3	17.3	92.9	21.6	85.3	36.6
13	—	—	—	—	—	—	—	—	140.5	23.8	—	—	—	—
14	—	—	—	—	59.8	47.7	25.9	24.7	88.4	13.4	—	—	—	—
15	—	—	—	—	—	—	96.7	35.7	—	—	134.6	36.6	182.0	103.0
16	—	—	—	—	179.6	60.4	37.8	23.9	—	—	116.9	22.6	109.4	56.9
17	—	—	—	—	—	—	—	—	—	—	—	—	—	—
18	—	—	—	—	56.4	38.9	—	—	116.1	13.9	40.3	11.8	53.8	40.0
19	—	—	—	—	—	—	58.2	32.9	—	—	134.9	44.2	—	—
20	—	—	—	—	86.1	43.9	30.1	19.2	—	—	52.4	12.5	—	—
21	—	—	—	—	—	—	—	—	—	—	—	—	—	—
22	—	—	—	—	—	—	20.9	18.9	79.1	9.7	42.0	5.9	—	—
23	—	—	—	—	—	—	93.1	38.7	—	—	—	—	—	—
24	—	—	—	—	—	—	56.4	16.4	—	—	133.2	20.8	—	—

9.4 Results for the Cluster Recognition Problem

Benchmark: Cluster Recognition Problem														
	B-BP		GR		DBD		EDBD		DA		QP		EQP	
	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR
01	54.8	40.7	—	—	52.3	40.0	28.4	20.3	96.7	20.8	38.6	22.4	—	—
02	40.4	31.6	100.8	73.2	16.3	10.8	15.0	13.0	13.8	10.1	10.5	7.5	22.3	18.8
03	49.9	34.8	216.4	105.6	71.7	56.5	70.1	27.3	24.1	18.1	53.1	32.0	—	—
04	23.5	19.1	79.8	71.2	14.6	12.0	14.8	13.0	11.7	10.1	9.7	8.4	22.5	16.9
05	53.0	45.8	—	—	36.0	27.0	23.8	21.2	91.6	17.6	17.0	13.2	52.8	43.5
06	39.8	30.5	98.4	81.0	13.7	11.4	15.5	14.2	12.4	10.1	8.9	7.2	21.8	19.6
07	55.4	32.7	211.0	103.5	65.2	36.3	61.1	23.7	48.8	17.3	64.5	20.1	103.2	60.0
08	24.3	20.1	83.6	67.5	14.4	12.0	15.4	13.3	11.7	9.3	9.8	7.8	22.7	17.8
09	62.2	49.0	56.7	50.3	67.6	46.2	46.4	39.0	30.2	15.6	53.5	34.9	—	—
10	51.2	41.8	135.8	120.8	24.0	18.7	29.9	25.5	24.0	18.8	19.7	13.7	37.3	29.7
11	50.6	45.3	123.5	61.5	123.5	61.5	108.9	51.9	40.0	29.1	84.8	45.6	—	—
12	32.3	28.0	105.0	94.8	26.0	21.8	26.7	23.2	21.7	18.9	17.1	15.9	37.4	27.2
13	—	—	234.4	99.8	—	—	—	—	—	—	53.3	23.4	—	—
14	22.9	15.9	46.5	41.8	20.4	12.3	17.1	15.4	10.7	8.8	7.5	6.5	—	—
15	—	—	—	—	—	—	—	—	—	—	111.8	42.6	—	—
16	18.8	13.2	55.0	46.0	26.9	12.6	18.0	14.2	12.1	9.2	8.4	6.9	133.8	80.6
17	—	—	—	—	—	—	—	—	—	—	46.0	12.3	—	—
18	19.0	14.4	47.6	40.1	17.7	11.4	16.1	15.1	11.0	9.7	7.7	6.7	—	—
19	—	—	—	—	—	—	130.6	70.4	—	—	110.8	35.8	—	—
20	24.0	15.7	50.3	41.8	23.6	12.3	17.5	14.6	12.5	9.7	8.2	7.2	128.7	54.7
21	—	—	239.0	110.5	—	—	—	—	—	—	53.7	36.5	—	—
22	27.1	19.3	71.7	64.5	56.9	21.1	29.2	25.4	18.6	17.1	13.6	12.1	—	—
23	—	—	—	—	—	—	—	—	—	—	—	—	—	—
24	79.3	62.2	79.3	62.2	60.8	24.6	29.6	25.2	22.6	17.1	15.3	12.3	—	—

9.5 Results for the Correlated Cluster Recognition Problem

Benchmark: Correlated Cluster Recognition Problem														
	B-BP		GR		DBD		EDBD		DA		QP		EQP	
	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR	AR	FR
01	—	—	—	—	—	—	151.2	110.5	106.7	64.6	116.5	48.4	—	—
02	111.2	70.9	—	—	—	—	53.7	35.2	76.3	62.7	87.8	41.7	149.4	101.6
03	—	—	—	—	—	—	—	—	108.0	70.6	—	—	—	—
04	63.1	37.5	—	—	—	—	61.9	38.1	69.4	50.2	78.5	39.2	—	—
05	—	—	—	—	—	—	—	—	—	—	—	—	—	—
06	—	—	—	—	—	—	67.1	47.6	83.3	55.4	82.9	34.0	158.1	112.6
07	—	—	—	—	—	—	—	—	—	—	—	—	—	—
08	54.3	36.1	—	—	—	—	65.3	40.2	56.4	46.9	58.5	30.0	—	—
09	—	—	—	—	155.5	112.4	106.9	38.8	66.7	39.1	150.4	102.5	180.3	120.9
10	60.2	40.8	—	—	151.7	78.1	53.2	40.7	52.8	37.9	75.4	49.1	135.3	79.1
11	—	—	—	—	133.8	90.5	101.1	57.5	66.3	51.5	147.4	105.3	170.3	124.2
12	69.5	47.7	—	—	161.4	133.7	74.2	41.2	55.1	40.5	79.6	54.3	132.9	105.3
13	—	—	—	—	—	—	142.2	115.7	—	—	—	—	—	—
14	—	—	—	—	139.4	32.7	83.4	38.8	125.5	44.3	98.7	28.7	155.6	112.7
15	—	—	—	—	—	—	113.5	87.7	—	—	—	—	—	—
16	—	—	—	—	—	—	74.1	40.6	95.1	59.0	97.0	52.1	—	—
17	—	—	—	—	—	—	—	—	—	—	—	—	—	—
18	—	—	—	—	143.1	122.6	69.3	35.3	90.8	41.8	58.7	24.8	134.2	117.3
19	—	—	—	—	—	—	—	—	—	—	—	—	—	—
20	—	—	—	—	—	—	81.7	35.2	98.7	58.7	71.4	23.9	158.7	136.4
21	—	—	—	—	—	—	134.1	89.6	149.7	60.7	—	—	—	—
22	—	—	—	—	145.6	81.8	79.2	35.2	84.4	38.5	133.5	32.1	102.5	57.6
23	—	—	—	—	—	—	153.4	84.5	—	—	—	—	—	—
24	—	—	—	—	—	—	110.5	62.9	117.3	51.5	134.0	46.3	136.9	86.0

References

[AS91a] L. B. Almeida and F. M. Silva. A distributed solution for data orthonormalisation. In: T. Kohonen, K. Mäkisara, O. Simula and J. Kangas, editor, *Artificial Neural Networks*, pp. 943–948, Amsterdam, 1991. North Holland.

[AS91b] L. B. Almeida and F. M. Silva. Speeding-up backpropagation by data orthonormalisation. In: T. Kohonen, K. Mäkisara, O. Simula and J. Kangas, editor, *Artificial Neural Networks*, pp. 1503–1507, Amsterdam, 1991. North Holland.

[AS92] L. B. Almeida and F. M. Silva. Adaptive decorrelation. In: I. Aleksander and J. Taylor, eds., *Artificial Neural Networks*, pp. 149–156, Amsterdam, 1992. North Holland.

[BH69] A. E. Bryson and Yu-Chi Ho. *Applied Optimal Control.* Blaisdell, New York, 1969.

[BH92] K. Balakrishnan and V. Honavar. Improving convergence of backpropagation by handling flat spots in the output layer. In: I. Aleksander and J. Taylor, eds., *Artificial Neural Networks*, pp. 1003–1009, Amsterdam, 1992. North Holland.

[BlC88] S. Becker and Y. le Cun. Improving the convergence of backpropagation learning with second order methods. In: D. Touretzky, G. Hinton, and T. Sejnowski, eds., *Proceedings of the '88 Connectionist Models Summer School*, pp. 29–37. Carnegie-Mellon-University, 1988.

[Fah88] S. E. Fahlmann. Faster-learning variations on back-propagation. In: D. Touietzky and T. Sejnowski G. Hinton, eds., *Proceedings of the '88 Connectionist Models Summer School*, pp. 38–51. Carnegie-Mellon-University, 1988.

[FD92] M. Fombellida and J. Destine. The extended quickprop. In: I. Aleksander and J. Taylor, eds., *Artificial Neural Networks*, pp. 973–977, Amsterdam, 1992. North Holland.

[Hes80] M. R. Hestenes. *Conjugate Direction Methods in Optimization.* Springer, New York, 1980.

[HKP91] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley Publishing Company, Redwood City, 1991.

[HN89] R. Hecht-Nielsen. Theory of the back propagation neural network. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 2, pp. 381–386, 1989.

[HN91] R. Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, Reading, 1991.

[HS88] D. R. Hush and J. M. Salas. Improving the learning rate of back-propagation with the gradient reuse algorithm. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 1, pp. 441–447, 1988.

[Jac88] R. A. Jacobs. Increased rates of convergence through learning rate adaption. *Neural Networks*, 1:295–307, 1988.

[Jud87] S. Judd. Learning in networks is hard. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 2, pp. 685–692, June 1987.

[KDS90] A. Krzyzak, W. Dai, and C. Y. Suen. Classification of a large set of handwritten characters using modified backpropagation model. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 3, pp. 225–232, 1990.

[Lue73] D. G. Luenberger. *Introduction to linear and nonlinear Programming.* Addison-Wesley Publishing Company, London, 1973.

[MP88] M. L. Minsky and S. Papert. *Perceptrons.* MIT Press, Massachusets, 1988.

[MW90] A. A. Minai and R. D. Williams. Backpropagation heuristics: A study of the extended delta-bar-delta algorithm. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 1, pp. 595–600. 1990.

[Par85] D. B. Parker. Learning logic. Technical Report 47, Center for Computational Research in Economics and Management Science, MIT, 1985.

[PNH86] D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on learning by back-propagation. Technical Report CMU-CS-86-126, Carnegie-Mellon-University and Comp. Science. Dep, Pittsburgh PA, 1986.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In: D. E. Rumelhart and J. McClelland, eds., *Parallel Distributed Processing.* MIT Press, 1986.

[Roj93] R. Rojas. *Theorie der Neuronalen Netze – Eine systematische Einführung.* Springer, Berlin, March 1993.

[Sat91] A. Sato. An analytical study of the momentum term in a backpropagation algorithm. In: T. Kohonen, K. Mäkisara, O. Simula and J. Kangas, editor, *Artificial Neural Networks*, pp. 617–753, Amsterdam, 1991. North Holland.

[SH87] W. S. Stornetta and B. A. Hubermann. An improved three-layer, back propagation algorithm. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 2, pp. 637–643, June 1987.

[SR87] T. Sejnowski and C. Rosenberg. Netalk: A parallel network that learns to read aloud. *Computer Systems*, 1:145–168, 1987.

[Wat87] R. L. Watrous. Learning algorithms for connectionist networks: Applied gradient methods of nonlinear optimization. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 2, pp. 619–627, June 1987.

[Sal92] R. Salomon. *Verbesserung konnektionistischer Lernverfahren, die nach der Gradientenmethode arbeiten.* PhD thesis, TU Berlin, November 1992.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science.* PhD thesis, Harvard University, November 1974.

[Wer88] P. J. Werbos. Backpropagation: Past and future. In: *Proceedings of the IEEE 1st International Conference on Neural Networks*, vol. 1, pp. 343–353, 1988.

[WSW87] R. L. Watrous, L. Shastri, and A. Waibel. Learned phonetic discrimination using connectionist networks. In: *European Conference on Speech Technology*, pp. 377–380, Edinburgh. 1987.